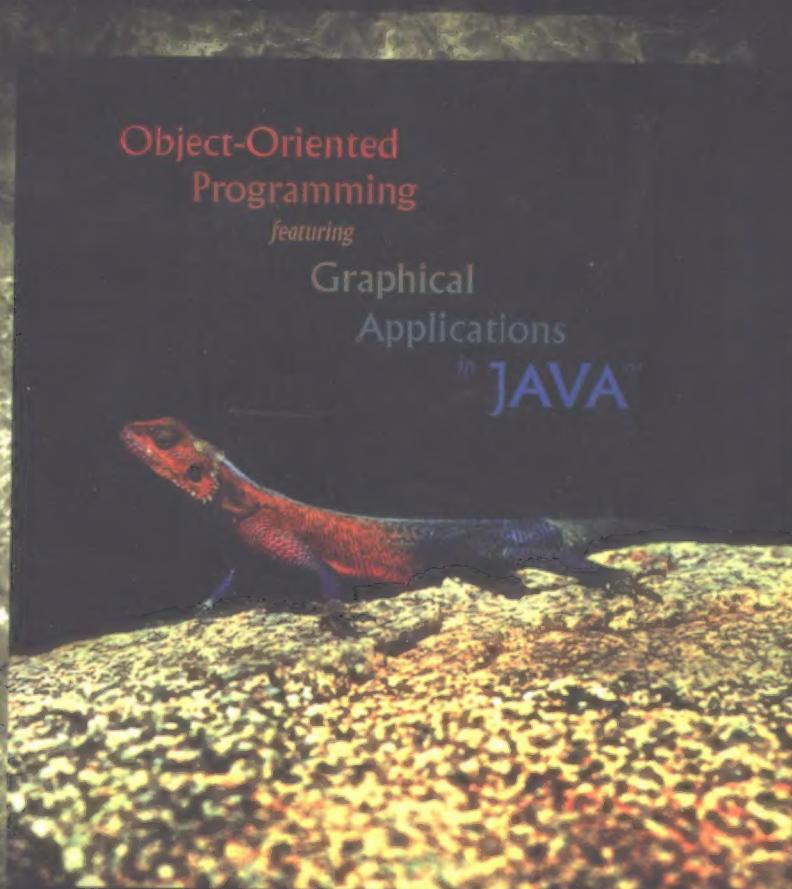


计 算 机 科 学 从 书

面向对象程序设计

—— 图形应用实例

(美) Michael J. Laszlo 著 杨秀梅 何玉洁 牟永敏 周冬梅 等译



Object-Oriented Programming
Featuring Graphical Applications in Java



机械工业出版社
China Machine Press



在现代软件的实际开发中，理解面向对象程序设计(OOP)的思想和方法至关重要。本书清晰地论述了面向对象程序设计的核心原理，并用大量实例加以阐释（其中许多是二维图形应用程序）。读者只要有Java基本知识就可看懂本书，无须二维图形程序接口Java 2D和统一建模语言UML的概念。在使用这些知识前，书中先进行了详细的介绍。

本书特色

- 面向对象的重要概念：使用Java的二维图形应用程序接口Java 2D帮助理解这些概念
- 两大类交互程序实例：基于文本命令和基于Swing的图形用户接口程序
- UML子集：应用UML子集描述程序设计
- 更进一步的练习：联系理论和实际应用
- 设计模式：主要利用图形程序设计介绍迭代器、模板方法和组合设计模式
- 面向对象的程序框架：基于AWT和Swing创建使用GUI的程序
- 类和接口包：提供60多个类和接口用于创建和绘制二维图形
- 在线代码和文档：便于下载

作者简介

Michael J. Laszlo，获美国普林斯顿大学计算机科学博士学位，主攻计算机图形学和计算几何学，现为美国Nova东南大学计算机科学系教授。



ISBN 7-111-10143-X



9 787111 101437

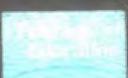


华章图书

网上购书：www.china-pub.com

北京市西城区百万庄南街1号 100037
购书热线：(010)68995259, 8006100280 (北京地区)
总编信箱：chiefeditor@hzbook.com

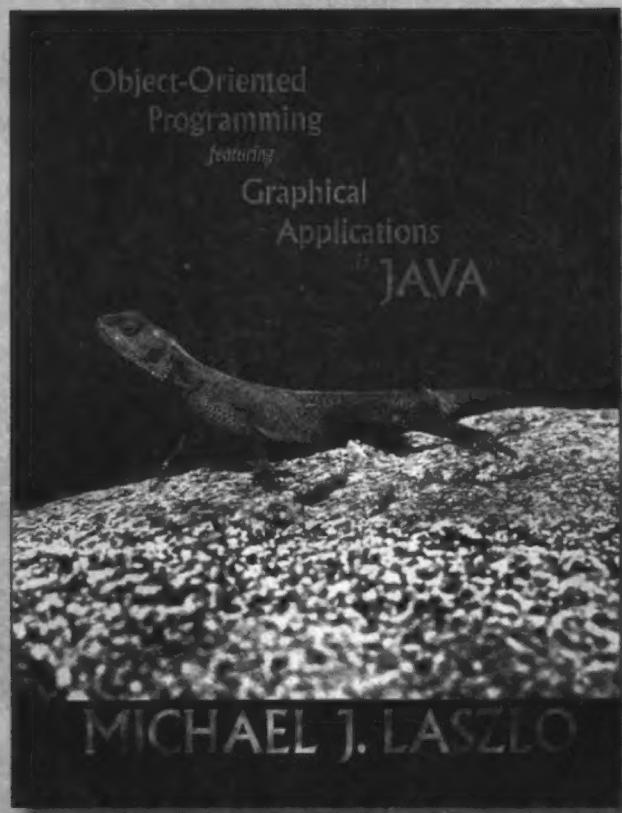
ISBN 7-111-10143-X/TP · 2401
定价：35.00 元



计 算 机 科 学 丛 书

面向对象程序设计 ——图形应用实例

(美) Michael J. Laszlo 著 杨秀梅 何玉洁 卞永敏 周冬梅 等译
Nova 东南大学



Object-Oriented Programming
Featuring Graphical Applications in Java

 机械工业出版社
China Machine Press

面向对象程序设计（OOP）的思想和方法在现代软件设计中越来越重要。本书使读者站在软件工程的高度，理解和掌握面向对象程序设计技术并能应用它解决实际问题。书中以大量的Java程序（大多数是二维计算机图形程序）为实例阐明了面向对象程序设计中的重要概念和设计方法。开篇先阐述了OOP中的对象模型、过程抽象和数据抽象，接着介绍了继承和组合，最后讨论了设计模式和应用程序框架。本书还使用了统一建模语言UML来描述一些设计概念，使读者站在更高的分析与设计层次来认识和理解所需解决的问题。本书还附有大量的练习，针对每节的内容提出问题，让读者进一步巩固所学的理论和方法。

本书可作为计算机专业本科生的教学参考，对涉及OOP的广大软件开发设计者而言也是不错的指导。

Simplified Chinese edition copyright © 2002 by PEARSON EDUCATION NORTH ASIA LIMITED and CHINA MACHINE PRESS.

Original English language title: Object-Oriented Programming Featuring Graphical Applications in Java, 1st ed. by Michael J. Laszlo, Copyright © 2002.

All rights reserved.

Published by arrangement with the original publisher, Pearson Education, Inc., publishing as Addison Wesley.

This edition is authorized for sale only in the People's Republic of China (excluding the Special Administrative Region of Hong Kong and Macau).

本书简体中文版由Pearson Education North Asia Ltd.授权机械工业出版社在中国大陆境内独家出版发行，未经出版者许可，不得以任何方式抄袭、复制或节录本书中的任何部分。

本书封面贴有Pearson Education培生教育出版集团激光防伪标签，无标签者不得销售。
版权所有，侵权必究。

本书版权登记号：图字：01-2002-1836

图书在版编目（CIP）数据

面向对象程序设计——图形应用实例 / (美) 拉斯洛 (Laszlo, M. J.) 著；杨秀梅等译。-北京：机械工业出版社，2002.7

（计算机科学丛书）

书名原文：Object-Oriented Programming Featuring Graphical Applications in Java
ISBN 7-111-10143-X

I. 面… II. ①拉… ②杨… III. JAVA语言－程序设计 IV. TP312

中国版本图书馆CIP数据核字（2002）第021162号

机械工业出版社（北京市西城区百万庄大街22号 邮政编码 100037）

责任编辑：杨海玲

北京第二外国语学院印刷厂印刷·新华书店北京发行所发行

2002年7月第1版第1次印刷

787mm×1092mm 1/16 · 21.5印张

印数：0 001-4 000册

定价：35.00元

凡购本书，如有倒页、脱页、缺页，由本社发行部调换

出版者的话

文艺复兴以降，源远流长的科学精神和逐步形成的学术规范，使西方国家在自然科学的各个领域取得了垄断性的优势；也正是这样的传统，使美国在信息技术发展的六十多年间名家辈出、独领风骚。在商业化的进程中，美国的产业界与教育界越来越紧密地结合，计算机学科中的许多泰山北斗同时身处科研和教学的最前线，由此而产生的经典科学著作，不仅擘划了研究的范畴，还揭橥了学术的源变，既遵循学术规范，又自有学者个性，其价值并不会因年月的流逝而减退。

近年，在全球信息化大潮的推动下，我国的计算机产业发展迅猛，对专业人才的需求日益迫切。这对计算机教育界和出版界都既是机遇，也是挑战；而专业教材的建设在教育战略上显得举足轻重。在我国信息技术发展时间较短、从业人员较少的现状下，美国等发达国家在其计算机科学发展的几十年间积淀的经典教材仍有许多值得借鉴之处。因此，引进一批国外优秀计算机教材将对我国计算机教育事业的发展起积极的推动作用，也是与世界接轨、建设真正的世界一流大学的必由之路。

机械工业出版社华章图文信息有限公司较早意识到“出版要为教育服务”。自1998年始，华章公司就将工作重点放在了遴选、移译国外优秀教材上。经过几年的不懈努力，我们与Prentice Hall, Addison-Wesley, McGraw-Hill, Morgan Kaufmann等世界著名出版公司建立了良好的合作关系，从它们现有的数百种教材中甄选出Tanenbaum, Stroustrup, Kernighan, Jim Gray等大师名家的一批经典作品，以“计算机科学丛书”为总称出版，供读者学习、研究及收藏。大理石纹理的封面，也正体现了这套丛书的品位和格调。

“计算机科学丛书”的出版工作得到了国内外学者的鼎力襄助，国内的专家不仅提供了中肯的选题指导，还不辞劳苦地担任了翻译和审校的工作；而原书的作者也相当关注其作品在中国的传播，有的还专诚为其书的中译本作序。迄今，“计算机科学丛书”已经出版了近百个品种，这些书籍在读者中树立了良好的口碑，并被许多高校采用为正式教材和参考书籍，为进一步推广与发展打下了坚实的基础。

随着学科建设的初步完善和教材改革的逐渐深化，教育界对国外计算机教材的需求和应用都步入一个新的阶段。为此，华章公司将加大引进教材的力度，在“华章教育”的总规划之下出版三个系列的计算机教材：针对本科生的核心课程，剔抉外版菁华而成“国外经典教材”系列；对影印版的教材，则单独开辟出“经典原版书库”；定位在高级教程和专业参考的“计算机科学丛书”还将保持原来的风格，继续出版新的品种。为了保证这三套丛书的权威性，同时也为了更好地为学校和老师们服务，华章公司聘请了中国科学院、北京大学、清华大学、国防科技大学、复旦大学、上海交通大学、南京大学、浙江大学、中国科技大学、哈尔滨工业大学、西安交通大学、中国人民大学、北京航空航天大学、北京邮电大学、中山大学、解放军理工大学、郑州大学、湖北工学院、中国国家信息安全测评认证中心等国内重点大学和科研机构在计算机的各个领域的著名学者组成“专家指导委员会”，为我们提供选题意见和出版监督。

权威的作者、经典的教材、一流的译者、严格的审校、精细的编辑，这些因素使我们的图

书有了质量的保证，但我们的目标是尽善尽美，而反馈的意见正是我们达到这一终极目标的重要帮助。教材的出版只是我们的后续服务的起点。华章公司欢迎老师和读者对我们的工作提出建议或给予指正，我们的联系方法如下：

电子邮件：hzedu@hzbook.com

联系电话：(010) 68995265

联系地址：北京市西城区百万庄南街1号

邮政编码：100037

专家指导委员会

(按姓氏笔画顺序)

尤晋元	王 珊	冯博琴	史忠植	史美林
石教英	吕 建	孙玉芳	吴世忠	吴时霖
张立昂	李伟琴	李师贤	李建中	杨冬青
邵维忠	陆丽娜	陆鑫达	陈向群	周伯生
周克定	周傲英	孟小峰	岳丽华	范 明
郑国梁	施伯乐	钟玉琢	唐世渭	袁崇义
高传善	梅 宏	程 旭	程时端	谢希仁
裘宗燕	戴 葵			

译 者 序

在现代软件开发中，理解面向对象程序设计(OOP)的思想和方法非常重要。虽然有很多程序设计语言都支持OOP，如Smalltalk、C++、Java等，但一个初学者往往是在学习Java时开始接触这种思想。Java语言对OOP思想进行了完整、清晰的表达描述，它是一种纯粹的面向对象语言。因此在本书中用Java语言把丰富但抽象的OOP思想和具体示例结合起来，用具体的示例来帮助大家更快、更直观地理解这种思想和方法。同时这些示例大都为二维图形程序，不仅能学到如何进行图形程序设计，而且在学习中会感到生动有趣。

本书由浅入深，从基本的面向对象概念到软件重用方法逐步进行了介绍。第1章到第3章阐述了OOP中的对象模型、过程抽象和数据抽象等基本概念；第4章和第5章介绍简单的类重用方法——继承和组合；最后第6章和第7章讨论设计重用方法——设计模式和应用程序框架。

尤其值得强调的是：本书使用了统一建模语言UML来描述其中的一些设计概念，这可以使我们站在更高的分析与设计层次来认识、理解解决问题的方法；同时，本书引入了设计模式，这将会使我们在工作中受益于前人的设计成果。

本书中还附有大量的练习，它们都是在每节的基础上提出问题，让大家更进一步理解所学理论和方法。

读者只需要有Java的基本知识就可以看懂本书，其他新的概念如UML、Java 2D等在使用前都会有详细介绍。

本书主要译者为：杨秀梅、何玉洁、牟永敏、周冬梅。参加本书翻译和其他工作的还有马爱华、李少波、朱丹、王帅、贺莹、丁凌云、成然、杨永刚、柏亮、邓涛、朱红松、刘泽深、杨雄高、迟玉强、刘惠华、崔仲凯、李龚、刘文娜、付立武、彭兰茜、李燕华、刘彬、李跃等，在此向他们表示感谢。

译 者
2002年1月

前　　言

本书主要目的是用Java语言来探究面向对象程序设计(OOP)的基本思想。对象模型把知识和行为封装在对象中，是面向对象程序设计的基础。由于对象模型对处理程序复杂性很有效而且有越来越多的程序设计语言提供对它的支持，如Smalltalk、C++、Java等，因而近年来，这种程序设计模型占据越来越重要的地位。

面向对象的程序设计方法最早出现在Simula和Smalltalk等语言中，并且已经延续了几十年，但是一个新手往往是在学习Java时开始接触对象模型这种思想。Java语言为表达面向对象程序设计思想提供了清晰的表示法，因此在本书中将会用Java语言把丰富但抽象的面向对象程序设计思想和具体示例结合起来，而且大家可以以Java为工具编写、编译、运行书中的Java程序。此外，本书的大部分示例和练习都是和二维(2D)计算机图形相关的，有的还能产生有趣和令人惊讶的图形效果。这里之所以要使用2D图形示例，一是为了吸引读者，二是为了把新的设计思想和实际应用相结合。随着内容的深入，新的东西将不断被补充到2D图形示例的面向对象图形的类和接口中。

本书适合于已具有Java基础知识的读者，如果你还没有相关知识，请先阅读相关Java语言方面的书籍。尽管本书的目的不是讲授Java语言，但几乎所有的Java基本特性都在使用前进行解释。本书还依赖于两个附加的资源：一是Java 2中的Java 2D应用程序接口(Java 2D API)，用于产生二维图形；二是用于表示系统设计的统一建模语言(UML)的一个小子集，其中主要是用类图来表示系统的静态结构，用顺序图来表示对象间交互。Java 2D API和UML的特征将在需要的时候进行介绍，你不需要事先熟悉。

内容组织

第1章介绍对象模型的基本概念：对象和类，消息传递和方法，以及软件重用的四个基本机制——组合、继承、设计模式、应用程序框架。在其他重要的程序设计模型方面，要介绍对象模型。

第2章讨论了过程抽象，而过程被看作具体实现被隐藏的操作。本章中还讨论了Java的异常处理机制和基于过程抽象的两个标准程序设计技术：过程分解(即过程是由其他操作定义的)和递归(即过程是由过程实现的操作自身定义的)。之所以在本书的开头就讨论过程的概念，是因为过程在对象模型中起着很重要的作用。

第3章讨论了数据抽象。数据抽象是将数据值的具体内部结构隐藏，而把它看作一组相关操作和一个使用这些操作的协议。在对象模型中，数据值被视为对象。本章中还讨论了封装(相关软件元件组合在一起的技术)和信息隐藏。最后介绍了Java中的2D计算机图形，并用Java语言开发了一个计算机图形应用程序模板。

第4章讨论了软件重用的主要机制——组合。应用组合方法，可以将一个新类定义为由其他类组合成的类，这些类称为新类的组件。组合类的每个实例都包含了自己的组件。本章最后给出了一个交互的计算机图形应用程序的模板，读者可以利用它编写用户实时交互程序。

第5章讨论了继承，继承是指新类可以获得已有类（称为父类）的属性和行为，这样称新类是已有类的子类。在本章中讨论了三种基本的继承方法：扩展继承（子类是在继承父类的属性或行为的基础上增加新的属性或行为）；特征继承（子类重新定义了一个或多个它所继承的行为）；说明继承（子类实现了父类定义但没有实现的行为）。在本章中还介绍了如何使用继承来创建相关联的一组数据类型，同时介绍了多态性。

第6章讨论了设计模式。设计模式是描述了解决设计过程中反复出现的问题的有效方法，主要包括解决问题的一组软件元件和如何组合这些软件元件。在众多的设计模式中，本章只讨论三种：迭代器模式、模板方法模式、组合模式。迭代器模式提供访问一个聚集的各个元素的方法，而又隐藏其内部结构。模板方法模式定义一个由具体步骤和抽象步骤组成的算法，使得它的子类通过实现抽象步骤就可以使算法“新生”。组合模式用于将对象组合成基本组件和组合体的层次结构，使客户对基本组件和组合体能进行统一处理。本章应用这三种设计模式设计了多个图形应用程序，如有限点集三角形剖分，建立构造区域几何图形（Constructive Area Geometry, CAG）树（由布尔集合运算并、交和差把2D图形组合成的二叉树），以及构造情景图（scene graphic）（由基本组件和复合图形构成的层次图）。（尽管这样简单描述会使你觉得这些图形程序例子深奥而复杂，但配合上图形，你会发现它们很容易理解。）最后还介绍了其他的设计模式以及设计模式分类的标准方案。

第7章主要讲述了应用程序框架。应用程序框架主要是为了简化在某个特定领域的应用程序的设计。程序员只要依据应用程序框架的协定扩展和实现应用程序框架中提供的类和接口就可以很容易地开发出自己的应用程序。在Java中，应用程序框架是由AWT（Abstract Window Toolkit）、Swing和Java的事件模型组成的，这三部分组合起来构成创建图形用户接口（GUI）应用程序的框架。本章最后一部分是开发基于GUI的绘制和编辑各种图形的程序。

如何使用本书

练习是本书的重要部分，并且是紧随着讲解的内容出现的。有些练习要求大家实现刚刚介绍过的类，有些要求大家使用新介绍的概念和类来设计程序。总之本书中的大部分资料是不断累积的，后面章节中可能用到前面定义的类，很多类和接口被不断加入到面向图形的程序包中。有时，为了理解使用新的概念，会在后面部分修改前面定义过的类。

补充资料

本书中的练习包括从简单的、很容易回答的问题到独立的程序设计，以及非常复杂的程序设计项目。特别重要的练习或是介绍书中后面需用材料的练习都标明为重点，对这些练习，如果你不想解决它们，至少应该了解其中一部分重要内容。

本书中所有的Java程序（按章排列）和图形包都可以到<http://www.aw.com/cssupport>地址下载，该站点中还可以找到有关如何下载和安装这些文件的帮助资料。书中所有的图形的PowerPoint幻灯片文档也在该站点中。书中练习的答案面向教师提供，需要的教师可以和Addison Wesley Longman的销售代表处联系。

致谢

Nova东南大学计算机信息和科学研究生院（SCIS）为这本书的写作提供了机会和良好的

环境。特别要感谢SCSI的院长Edward Lieblein和我的同事，尤其是Maxine Cohen、Sumitra Mukherjee和Junping Sun，他们给了我很大鼓励和支持。

很荣幸有机会在SCSI从事与本书内容相关的教学工作，如面向对象程序设计、程序设计语言理论和计算机图形学。本书大部分材料都是在教学工作中积累起来的。我还要感谢那些提出问题和见解的学生，虽然至今我叫不出他们许多人的名字。最后要感谢数学与计算机科学研究所 (Institute for Mathematics and Computer Science, IMACS)的同事和学生。

我很感激下列的审阅者，他们花费大量的时间和精力撰写审阅意见，我们最后的手稿采用了其中的许多建议，他们是D. Robert Adams (大峡谷科罗拉多州立大学), Manuel E. Bermudez (佛罗里达大学), James R. Connolly (奇科加利福尼亚州立大学), Frank Coyle (南查珀尔希尔北卡罗来纳大学), John R. Glover (休斯顿大学), Chung Lee (加利福尼亚州立大学波蒙纳分校), Ronald McCarty (宾夕法尼亚伊利市贝伦德学院), Jong-Min Park (圣迭戈加利福尼亚州立大学), Shih-Ho Wang (戴维斯加利福尼亚大学) 和Marvin V. Zelkowitz (马里兰大学)。

我还要感谢Addison Wesley的本书编辑Maite Suarez-Rivas女士的真诚支持。同时对Jarrod Gibbons (市场协调员), Gina Hagen (设计经理), Katherine Harutunian (项目编辑), Michael Hirsch (营销经理), Marilyn Lloyd (项目经理) 和Patty Mahtani (助理管理编辑) 表示深深的感谢。

最后，我要感谢我的家人：我的妻子Elisa和我们的孩子Arianna Hannah和David Joshua, 还有我的父母Maurice 和Phyllis。他们的爱、鼓励和耐心是我完成这本书的动力。

目 录

出版者的话	
专家指导委员会	
译者序	
前言	
第1章 对象模型	1
1.1 对象模型概念	2
1.1.1 对象	2
1.1.2 消息	3
1.1.3 对象接口	4
1.1.4 方法和过程	5
1.1.5 封装	6
1.1.6 类和对象实例化	7
1.1.7 类和接口	9
1.1.8 关联	9
1.1.9 组合	11
1.1.10 继承	12
1.1.11 设计模式与程序设计框架	14
1.2 对象模型和其他程序设计模型	15
第2章 过程抽象	19
2.1 抽象操作和过程	19
2.2 过程说明	22
2.3 异常	26
2.3.1 受检查异常和不受检查异常	27
2.3.2 抛出异常	28
2.3.3 捕捉异常	29
2.3.4 处理异常	29
2.3.5 使用异常	31
2.4 过程分解	32
2.5 递归	37
小结	43
第3章 数据抽象	44
3.1 抽象数据类型	44
3.2 说明和实现数据抽象	45
3.2.1 点	46
3.2.2 矩形	55
3.3 封装	60
3.3.1 封装和类定义	61
3.3.2 信息隐藏	62
3.4 Java图形基础	64
3.4.1 Java 2D API绘图模型	64
3.4.2 获取绘图环境	65
3.4.3 创建图形对象	67
3.4.4 设置绘图环境的属性	67
3.4.5 绘图	69
3.5 Java图形程序实例	70
3.5.1 画矩形	70
3.5.2 图形程序模板	72
小结	74
第4章 组合	75
4.1 组合和聚集	75
4.2 随机数生成器	76
4.2.1 Java的Random类	77
4.2.2 随机整数	79
4.2.3 固定范围内的随机整数	82
4.2.4 随机点	84
4.2.5 随机矩形	89
4.2.6 画多个矩形	92
4.3 多组件组合	95
4.3.1 Java的Vector类	96
4.3.2 折线	98
4.4 表达一致性约束	104
4.4.1 概述	104
4.4.2 椭圆	106
4.4.3 有理数	111
4.5 交互图形程序	117
4.5.1 随机点	117

4.5.2 交互图形程序模板	121	6.4.1 组合图	219
小结	123	6.4.2 建立坐标轴	223
第5章 继承	125	6.4.3 可变换组合图	227
5.1 继承的使用	125	6.4.4 组合模式的结构和应用	237
5.2 扩展继承	128	6.5 设计模式分类	238
5.2.1 N步计数器	128	6.5.1 工厂方法模式	239
5.2.2 可变换点	130	6.5.2 适配器模式	240
5.2.3 直线	136	6.5.3 观察者模式	242
5.3 特化继承	139	6.5.4 策略模式	243
5.3.1 多边形	140	小结	244
5.3.2 标记计数器	145	第7章 面向对象应用程序框架	245
5.4 说明继承	146	7.1 用Java框架建立基于GUI的应用程序	245
5.4.1 接口和抽象类	146	7.1.1 框架的特点	245
5.4.2 矩形几何图形	148	7.1.2 Java的AWT和Swing	247
5.4.3 几何图形抽象	152	7.2 Java事件模型	248
5.5 多态性	158	7.2.1 概述	248
5.5.1 Java的多态性机制	158	7.2.2 创建点集程序	251
5.5.2 Java的Comparable接口与排序	161	7.2.3 编辑点集程序	256
5.5.3 替代原则	164	7.2.4 编辑多边形程序	260
5.6 Figure和Painter类	168	7.2.5 重设计编辑点集程序	262
5.6.1 图形	168	7.3 组件	267
5.6.2 填充和画图的绘图工具	170	7.3.1 Component和Container类	268
5.6.3 组合绘图工具	172	7.3.2 JComponent类	269
5.6.4 多边形绘图工具	176	7.3.3 JPanel类	269
小结	179	7.3.4 JButton类	270
第6章 设计模式	180	7.3.5 JLabel类	270
6.1 设计模式的重要性	180	7.3.6 JComboBox类	270
6.2 迭代器设计模式	181	7.3.7 JColorChooser类	271
6.2.1 Java的Iterator接口	181	7.4 布局管理器	272
6.2.2 动态多边形	184	7.4.1 流式布局	273
6.2.3 多边形迭代器	191	7.4.2 网格布局	274
6.2.4 迭代器模式的结构和应用	207	7.4.3 边界布局	274
6.3 模板方法设计模式	209	7.5 组件和事件监听器	275
6.3.1 布尔几何图形	209	7.5.1 处理颜色	275
6.3.2 半月图	212	7.5.2 记录颜色	277
6.3.3 构造区域几何图形	216	7.6 点集三角形剖分程序: Triangulate	281
6.3.4 模板方法模式的结构和应用	218	7.7 画图程序: DrawPad	288
6.4 组合设计模式	219	7.7.1 DrawPad的组件和图形管理器	288

7.7.2 DrawPad的事件监听器	295	附录B 图形程序框架	313
7.7.3 DrawPad的高亮度显示策略	303	附录C 统一建模语言UML符号概述	316
小结	307	附录D banana包结构	319
附录A 用户输入的读入和分析	309	参考文献	324

第1章 对象模型

众所周知，计算机处理的是二进制的0和1。为了底层处理，如取指令、分析指令、加法运算和存储数据，计算机的确是用位进行操作的。但实际上，我们眼中的计算机是和二进制的0和1相去甚远的，我们可以利用它实现想要的任何功能：在屏幕上画图、在窗口中输入文本、用鼠标点击选择一个对象，甚至使用强大的图形程序进入一个虚拟的三维世界。我们通过抽象的层次设计脱离与计算机底层处理的联系，以增强我们思考和表达的能力，而隐藏计算机工作的内部细节。当今的软件将计算机转变成定义自己的运算规则而且几乎与计算机内部处理没有明显联系的虚拟机。

抽象的概念不仅在使用计算机时用，而且在编写计算机程序时也使用。我们使用大多数计算机语言提供的高级特性进行程序设计，例如迭代、递归、条件表达式和过程调用，而不使用计算机本身的语言（机器语言）。这样程序员就可以避开计算机底层逻辑的细节和繁杂，而致力于应用抽象方法思考如何恰当地解决面对的问题。如果没有这样的抽象，程序员的效率将被限制在机器层次的位运算上，我们今天就不可能享受这么多令人难以置信的应用软件带来的方便和快乐。

什么是抽象？抽象是事物的一个理想化的模型，这个模型体现了事物的本质特征，而忽略了那些无关紧要的部分。抽象可以帮助我们处理事物的复杂性。通过抽象，我们可以更容易通过事物的本质特征来使用或理解事物，而避开事物的不重要部分的影响。我们利用抽象来处理复杂性。举例来说，在开车时，驾驶员将汽车作为具有如下要素的和对自己非常重要的一种抽象：加速用的油门，减速用的刹车、控制方向用的方向盘、播放音乐的收音机等，而其他的设备如给油设备、轮胎、打火栓和轴承不是驾驶员的抽象的内容，因为他不会直接操作这些设备。驾驶员不必关心不同汽车的不同的内部结构，无需接受驾驶不同汽车的培训，就可以轻易驾驶各种各样的汽车。

另一个例子，Java中的对象是一个抽象。对象是用来表示现实世界中的真实事物的（其中的真实是非常广泛的），但对象不可能也没有必要将真实事物的方方面面都描述和表示，对象只需表达真实事物在整个系统中那些本质的特性。如在一个学生注册系统中，每一个学生都用一个对象表示，系统可能需要学生对象的姓名或地址，修改学生对象的电话号码，或通知它已经通过某一门课程的选修申请。实际上，学生对象仅能表示它所代表的真正学生的一小部分，这样的对象不能吃比萨、跳舞、读书，而只能产生学生注册系统所需要的行为。

面向对象程序设计语言像Java等都提供了创造和使用不同的抽象的特性。这些语言特性支持一系列的基本原则，它们组合起来就形成面向对象程序设计模型 (object-oriented programming model)，亦即通常所说的对象模型 (object model)。在这一章我们主要介绍的是对象模型。1.1节中主要介绍对象模型的基本概念。其他的程序设计模型，如强制程序设计模式和函数程序设计模型，支持其他类型的抽象并由其他的程序设计语言所实现。1.2节中将对象模型和Java语言与其他几种重要的程序设计模型结合起来介绍。

1.1 对象模型概念

Java程序将一组可以按预定的方式相互通信的对象组合在一起，完成一个预定的功能。没有一个单独的对象能满足需要，只有它们相互正确地联合才能完成系统的功能要求，每一个对象可提供其他对象要求的特定服务。当一个对象要求某个服务时，它会发一个请求（称为消息）给可提供服务的另一个对象，接受消息的对象通过执行它的操作来作出响应并常常要发送另外的消息给其他的对象。这样消息不断在对象组成的网上来回传送，这就是对象模型下完成计算的方式。我们想象中的Java程序运行时实际的对象也是这样工作的。

那么对象是从哪来的呢？每个对象的行为是如何描述的呢？对象之间又是如何协调工作的呢？所有这些都是由Java程序来完成的。Java程序是由一组类和接口的定义组成。每个对象都属于某个类。每个对象的行为都由它所属的类和所实现的接口决定。通过对对象的行为，它可以创建新的对象、取消已有的对象；同样通过对对象的行为，它可以执行一定的操作，并发送消息使其他对象也完成一定的操作。

本节余下部分将详细讨论对象模型中的基本元素——对象、消息、类、方法和各种各样的协作和重用，并概述这些基本元素是如何组合在一起的。

1.1.1 对象

对象（object）是一个可以接收和响应消息的软件元素。对象表示真实的事物，真实的事物的含义非常广泛。它可以是可触摸的东西，如飞机、苹果，也可以是抽象概念，如颜色和图形轮廓；它还可以是主动的、可以启动或控制过程的事物，如定时器、传感器或人，或是像电梯和字典一样只被动响应服务请求的事物；对象还可以是实现的制品，如链表中的节点，或是用户交互的按钮或菜单。正确地确立对象是程序设计中的一个重要问题，并常常和所涉及的问题有关。一般来说，在程序和程序实现过程中，那些在问题域中扮演着一定角色的事物都会被确立为对象。

对象的行为（behavior）是指如何响应它所接收的消息。事实上，对象能接收它能理解的对应于不同消息的预定义行为（defined behavior）。每一个消息都会由预定义的行为响应；一个对象包含一组预定义的行为，因此对象能理解的消息种类是由对象的类决定的。

对象接收到一个消息时，它的响应是由预定义的行为和对象的当前状态（state）共同决定的。一个对象可以有任意个实例域（instance field）或简称域（field）。对象的状态和存储在它的域中的值相对应。对象响应消息时，它的状态也随之变化，称为状态转变（state transition），这也就意味着它的域值的改变。对象的状态记录它从创建开始到现在的—切过程。换句话说，对象的状态捕获和它的行为相关的历史事件。

除了行为和状态，对象还有一个标识（identity）用来区别它和其他对象。标识不依赖于对象的状态，也不随对象状态的变化而变化。这就如同虽然你的状态变化了（年龄的增长，吃爆米花，明白了一个道理），但你并不会变成另一个人。如果同一个类中的两个对象恰巧状态相同，那么它们仍然是不同的对象。

到目前为止，我们分别定义了对象的三个特性：预定义行为、状态和标识。对象如何响应接受的消息是由预定义行为和它的状态共同决定的。对象要接收消息，那它就要和其他对象相区别——它必须要有自己的标识。

为了用Java说明这些概念，这里举例一个例子，我们将用一个类表示笛卡儿平面中的点(见图1-1)。类Point有两个域，分别表示点的x和y的坐标。语句将完成下列操作：

```
Point p = new Point(3, 2);
```

- 创建一个Point类的新对象。
- 初始化点的x坐标值为3，y坐标值为2。
- 声明一个新的引用变量p。
- 指定变量p为对新的Point对象的引用。

虽然简略，但上述语句却完成了创建对象、初始化它的状态并确定了对它的引用。接着就可以通过引用变量p或这个引用的副本发送消息给这个新Point对象。

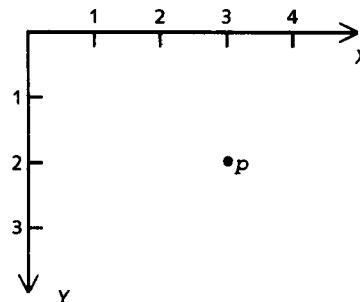


图1-1 平面和点 $p=(3, 2)$

1.1.2 消息

对象间通过相互发送消息(message)来进行交互。发送消息的对象称为发送者(sender)或客户(client)，而接受消息的对象称为接收者(receiver)或服务器(server)。这里用客户-服务器这个术语来表达，因为发送者请求另一个对象的服务时被视为客户，提供服务的接收者被视作服务器。为了完成所提供的服务，一个对象通常还会依靠另外一些对象的服务，也就是说对象有时扮演服务器的角色，有时却扮演客户的角色。通常对象扮演的角色是和消息相关的：相对于收到的消息，对象为响应服务的服务器；但为了完成请求服务，它又发送消息给其他对象，此时它又变成客户。

一个消息由下列三个元素组成：

- 消息名，又称选择器。
- 零个或多个参数列表，为接收对象提供数据信息。
- 指示接收对象的引用。

参数可能是对象引用或是基本数据类型(如整型)值。注意，尽管发送者可以把自己作为参数传递，但消息中并不包括对发送者的引用。

你或许对Java消息的语法结构很熟悉。假设p是表示平面上一个点对象，语句：

```
p.setCoordinates(8, 9);
```

是一个消息，它由选择器setCoordinates、两个参数8和9和接收对象p组成。消息响应的结果是p把它的位置变为(8, 9)。

是哪个对象发送`p.setCoordinates(8, 9)`这个消息的呢？在这个简单的例子里不能确定。实际上，这个消息作为一条语句出现在某个对象的预定义行为（方法）体内，在执行这个方法的过程中，这个对象发送的消息。

通常一个对象还会发送消息给它自己。这种情况下，消息的接收者也就是发送者，消息的接收者可以从消息中省略掉。这样点`p`就会用语句

```
setY(4);
```

改变它的`y`坐标值为4。这个消息含有消息名`setY`和参数4，接收对象也就是发送者是隐含的。因为关键字`this`总是用来表示消息的发送者（它的代码正在执行中），所以上面的消息可以写成

```
this.setY(4);
```

使得接收者是显式的。

关键字`this`还可以作为参数，使对象可以发送一个标识自身为发送者的消息。例如，如果`p`和`q`都是`Point`对象，`p`可以发送消息

```
q.setCoordinates(this);
```

告诉`q`修改它的坐标值使它们和`p`的值相等。接收者`q`期望`setCoordinates`这一消息的参数是一个`Point`对象，事实上`this`就是一个`Point`对象，是用关键字`this`标识的发送者自己，即点`p`。

当对象收到一个消息时，它就按照预定义的行为完成响应，这些行为是由接收对象的方法（method）定义的，关于方法我们将稍后讨论。执行预定义行为只可能会产生三种结果：

- 返回一个值给消息发送者。
- 接收者状态的改变。
- 作为参数传给接收者的对象的状态变化。

所产生的这些结果是由特定的行为决定的，可能是一种、两种或者三种同时产生。有时接收者状态发生变化是由于它的一个或多个对象的属性的状态发生变化而引起的，如果这些属性正好被其他对象共享，那么不仅发送者和接收者，其他对象也会受到这种行为的影响。行为的结果有时很难理解，而且经常会带来意想不到的或不正确的行为的产生。

1.1.3 对象接口

对象可响应的消息是由对象接口决定的。对象的接口是以一组操作的形式出现的，每一个操作都对应于在响应某个消息时对象所完成的预定义行为。对象的接口是由它的类型（type）决定的，而对象的类型又是由它所属于的类决定的。

客户通过对对象的接口来理解对象支持的各种行为。例如，我们可以利用下面类似Java语言的语法来表达`Point`对象接口：

```
public class Point {
    // constructs a new point at position (x,y)
    public Point(int x, int y)

    // constructs a new point that is a copy of point p
    public Point(Point p)
```

```

// constructs a new point at (0,0)
public Point()

// returns the x coordinate of this point
public int getX()

// changes the x coordinate of this point to newX
public void setX(int newX)

// returns the y coordinate of this point
public int getY()

// changes the y coordinate of this point to newY
public void setY(int newY)

// changes the position of this point to (newX,newY)
public void setCoordinates(int newX, int newY)

// changes the position of this point
// to (p.getX(),p.getY())
public void setCoordinates(Point p)

// translates this point by dx along x and dy along y
public void moveBy(int dx, int dy)

// returns a string-descriptor for this point: "(x,y)"
public String toString()
}

```

使用对象操作是要遵守一定的规则，这些规则称为对象的协议（protocol），协议描述如何使用对象的每一个操作。客户要想和对象进行正确交互，就必须遵守对象的接口和协议。例如，当一个Point对象接收到带有一个参数的setCoordinates消息时，这个参数必须是Point对象；下面的消息就违反了对象协议：

```
p.setCoordinates(null);
```

这里讲的Point对象的接口相对比较简单，不过在本书后面，我们将看到有很多有趣的接口和协议的对象。

到目前为止，我们所描述的接口是指对象的公有接口（public interface），这里的公有强调的是任何类型的对象都可以使用它的操作，也就是说公有接口定义了一组可被任何对象发送的消息。除了公有接口外，对象还可以提供限制型的操作，供某些类型的客户使用。只供特定类型客户使用的操作称为限制型接口（restricted interface）。私有接口（private interface）是限制性最强的接口类型，而公有接口的限制最弱的。私有接口是公有接口的超集，它包括所有公有接口的操作，而且它还加入了一些仅对同一类的对象可用的私有操作。

Java提供另外两种限制型接口：保护型接口（protect interface）和包接口（package interface）。从公有接口到保护型接口，再到包接口，最后到私有接口，接口的访问限制越来越强，而作用会变得越来越大。接口访问权限的设置可以使一个授权的客户操纵一个对象而另一个未授权的客户则不能。

1.1.4 方法和过程

过程（procedure）是实现某一操作的一段代码。过程定义了一个可计算的程序，此程

序执行时完成过程所描述的动作。作为一个操作，过程获得输入、产生输出和副作用。过程通过一组参数来接收输入；过程把它的输出返回客户代码，常常是返回给调用它的过程；过程产生的副作用是指对象状态的变化和其他一些活动，像读输入和写输出等。

过程参数的数目和类型以及它的名字组合在一起称为过程的签名（signature）。例如，Point类中两个参数的方法setCoordinates

```
void setCoordinates(int newX, int newY)
```

的签名包含过程的名字setCoordinates和两个类型为int的参数表。方法

```
void setCoordinates(Point p)
```

却拥有不同的签名，因为它声明了一个Point类型的参数。注意过程的返回类型不是它签名的一部分。

方法（method）就是与一个对象有关的过程。当一个对象收到一个消息时，它就以执行它的某个方法来响应。这个消息的参数作为输入传递给该方法。

一个对象可以有任意数目的方法。当对象收到一个消息时，决定调用哪个方法来响应消息的过程称为方法绑定（method binding）。方法的选择是由消息元素——消息名和消息的参数数目和类型决定的。方法绑定就是找到签名和消息的元素相匹配的方法：方法名与消息名相匹配，方法的参数表与消息的参数表在类型和数目上相匹配。为了设定Point对象p的x和y坐标值，我们发送一个带有新坐标值的setCoordinates消息：

```
p.setCoordinates(3, 4);
```

在响应这个消息时，对象p执行具有签名

```
void setCoordinates(int newX, int newY)
```

如果在同一个对象中有两个或两个以上的方法具有相同的名字，称为方法名被重载（overloaded）。虽然它们名字相同，但它们的签名不同，用来区分它们是不同的方法。为了处理一个名字重载的消息，方法绑定把消息的参数表同候选方法参数表逐一进行比较，直到找到完成匹配的方法。在响应消息

```
p.setCoordinates(new Point(5, 6));
```

时，对象p执行具有签名

```
void setCoordinates(Point p)
```

的方法。这里方法名setCoordinates是重载的，所以利用消息的参数表来决定执行哪一个setCoordinates方法。

区分操作和方法是很重要的。操作描述的是一种行为，包括从输入到输出的映射以及所产生的副作用。这种行为是从调用操作的客户角度来看的。相反，方法是以过程的形式实现操作。操作描述了行为，而方法描述了操作如何实现。同样可以说操作是方法的抽象，操作展示方法实现的动作而隐藏了实现的过程。

1.1.5 封装

封装（encapsulation）是把一组相关软件元素组织到一起的方法。encapsulation这个词本意就是把一组元素包裹在一个外壳里。Java提供包来封装一组相关类，然而最重要的封

装形式是发生在单个对象级上：一个对象封装一组相关的数据和方法。

封装还可用于分隔构成对象的元素，使我们可以区分对象的接口和实现。如图1-2，我们可以把对象想像成一个表示它的公有接口的外包围层，而在外包围层里面可以嵌入零个或多个其他的包围层，代表逐渐增加限制的接口。

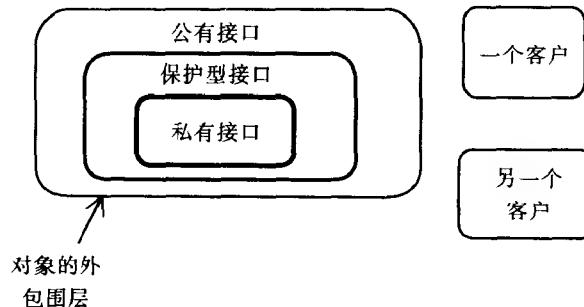


图1-2 对象封装一组接口

利用封装来隐藏那些不属于对象公有接口的软件元素称为信息隐藏 (information hiding)。Java提供了四个访问控制级别：公有 (public)、保护 (protected)，包 (package) 和私有 (private) ——用它们来区别哪些元素属于对象公有接口而哪些不属于，并指定隐藏那些不属于公有接口的元素的范围。

抽象和信息隐藏是相辅相成的概念。抽象揭示出了一个客户要使用一个对象必须需要了解的内容；信息隐藏则为了防止客户滥用对象，把它没有必要知道的内容隐藏起来。抽象和信息隐藏给出了对象提供的服务——对象接口，与此同时隐藏了是如何完成这些服务的——对象的实现。

1.1.6 类和对象实例化

每一个对象都属于某个类。类不仅决定了对象的类型，还决定它的域和方法。域和方法在类定义 (class definition) 中表达出来。创建一个新对象时，与对象类型相对应的类定义决定了对象的结构和行为。

下面以Point类为例来说明。到目前为止，我们已经看到很多关于Point对象的代码片段，并在1.1.3节对Point对象的预定义行为做了描述。基于以上知识，你可能已经明白如何使用点的一些操作，这也是实现Point类的基础。在Point类中，我们存储域中同名的x和y的坐标值，下面是类的定义：

```
public class Point {

    // fields
    protected int x, y;

    // constructors
    public Point(int newX, int newY) {
        x = newX;
        y = newY;
    }

    public Point(Point p) {
        this(p.getX(), p.getY());
    }
}
```

```

public Point() {
    this(0, 0);
}

// other methods
public int getX() { return x; }

public void setX(int newX) { x = newX; }

public int getY() { return y; }

public void setY(int newY) { y = newY; }

public void setCoordinates(int newX, int newY) {
    setX(newX);
    setY(newY);
}

public void setCoordinates(Point p) {
    setCoordinates(p.getX(), p.getY());
}

public void moveBy(int dx, int dy) {
    setCoordinates(getX() + dx, getY() + dy);
}

public String toString() {
    String res = "(" + getX() + "," + getY() + ")";
    return res;
}
}

```

因为所有的方法都声明为public，它们便组成了Point对象的公有接口；相反，x和y域声明为protected，所以它们属于对象的限制型接口。特别需要说明的是，只有和Point类在同一个包的类或是Point的子类对象才可以直接访问x和y域。在本书里，限制型接口的元素声明为protected。

用一个类生成一个新的对象的过程称为实例化 (instantiating)，产生的对象称为类的实例 (instance)。当用关键字new实例化一个类时，类的构造器就被调用执行。构造器调用为对象申请内存，创建类型(或与它的类建立联系)，并且初始化它的域值。构造器返回新对象的一个引用，之后，就可以通过这个引用或是这个引用的拷贝发送消息给对象。下面是一个例子：

```

Point p;
p = new Point(4, 2);
System.out.println("x: " + p.getX()); // x: 4

```

上面代码段创建了一个Point对象并初始化它表示点(4, 2)，并把这个新对象的引用赋给变量p，然后对象p收到消息getX，它返回x坐标值(4)并被打印出来。

一个对象可被多次引用。在下面的代码段里(接上代码段)，变量p的引用被拷贝到第二个变量q中：

```

Point q;
q = p;
q.setCoordinates(6, 8);
System.out.println("x: " + p.getX()); // x: 6
System.out.println("y: " + p.getY()); // y: 8

```

一旦`q=p`语句执行完，同一个`Point`对象有两个引用，分别存储在`p`和`q`中。`p`和`q`这两个变量指向同一个对象，最后三条语句的执行说明了这一点。语句

```
q.setCoordinates(6, 8);
```

通过存储在变量`q`中的引用改变了`x`和`y`坐标的值。最后两条语句通过存储在变量`p`中的引用取出对象的`x`和`y`坐标值并把它们打印出来。输出结果表明`p`引用的点通过`q`中的引用发送的消息`setCoordinates`改变了：变量`p`和`q`引用了相同的`Point`对象。在图1-3中，每个箭头的出发点是引用变量，箭头指向的是被引用对象。

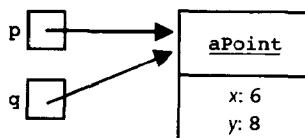


图1-3 `Point`对象被引用两次

1.1.7 类和接口

类有三个主要功能：第一，类定义了类型；第二，类提供了它的类型的实现，它定义了实例如何表示以及根据选定的表示方法如何实现；第三，类提供了初始化实例的构造器，如果是具体类，构造器用来创建和初始化新的实例。

Java中有两种类型的类：抽象类（`abstract class`）和具体类（`concrete class`）。两种类都定义了类型，它们的区别在于实现类型的程度不同：具体类提供了完整实现，而抽象类通常只提供一部分实现。抽象类声明的方法多于它实际实现的方法（那些没有实现的方法称为抽象方法（`abstract method`））。抽象类不可能实例化。事实是，如果一个抽象类的实例接收到一个与它的抽象方法相匹配的消息，那么它将无所适从，因为它不知道如何实现。

Java还提供了称为接口（`interface`）的结构，它定义一种没有提供任何实现的类型。接口中声明的所有方法都是隐含的抽象方法。同样，接口不能实例化。抽象类和接口都是用来作为其他类的超类。一个类扩展（`extend`）一个抽象类时，它就继承了抽象类的行为说明和它的部分实现。只有新类实现了它所继承所有抽象部分后，它才是具体类。一个类实现（`implement`）一个接口是通过实现接口声明的抽象方法完成的。只有新类在实现了接口声明的所有方法后，它才是具体类。

类是Java的基本元素，但这个概念在不同的面向对象程序设计语言中有不同的含义。在Smalltalk语言中，类是对象，要创建一个对象，只需向类发送一个消息告诉它将自己实例化。在Java中，类不是对象，但它却含有对象的一些特征。和对象一样，Java类有域（称为类域（`class field`）或静态域（`static field`））和方法（类方法（`class method`）或静态方法（`static method`））。和对象不同，类不是一个类型的实例。还有一些程序设计语言也使用类。Self语言是以原型为基础的语言，意思是说新的对象是通过拷贝已存在的对象而建立，然后再指定它们的不同之处。

1.1.8 关联

如果两个类A和B的实例之间可以发送消息，那么称A和B类之间存在一个关联

(association)。当对象a(A类)发送消息给对象b(B类)，那么就说这两个对象被链接(link)。要使对象a可以发送消息给对象b，对象a需要一个对象b的引用。最常用的方法就是在它的域中保存对对象b的引用。例如，一条直线段上有两个端点(Point对象)，则一个直线段对象包含两个域，每个域保存对一个端点对象的一个引用。如果要修改或查询端点的坐标值，直线段对象通过存储在这个域中的引用发送消息。

除了使用域，两个类之间的关联还能用其他方法实现。对象a可以创建一个类b的实例，只使用一次而没有保留对新对象的引用，这是一种方法。另外一个方法就是对象b作为消息的一部分(参数)发送给对象a。例如，一个Graphics2D对象(它定义用来绘制各种图形的方法)发送一个draw消息，参数是对要画的矩形的引用：

```
aGraphics2D.draw(aRectangle);
```

为了响应这个消息，aGraphics2D会向aRectangle发送消息来确定矩形的位置和大小。这样当draw方法执行的时候，对象aGraphics2D和aRectangle就被链接在一起了。

导航(navigability)指出了消息在两个链接对象之间的传递方向。例如，一条直线段存储了对它的两个端点的引用，但是这两个端点并不存储对这条线段的引用，这就是单向导航，直线段可以向它的两端点发送消息而反过来却不可以。双向导航也很常用。以学校注册系统里学生和课程之间的关联为例，一个学生对象保存了对他所选课程的引用，这样系统就可以追踪他的学习进度；同样，一个课程对象也保存了对选修本课程学生的引用，便于系统产生学生的名单、所在年级和类似的信息。学生对象和课程对象之间的链接是双向的——双方都可以向对方发送消息。

两个类之间的关联反映了两个概念之间的某种关系。一条线段有两个端点是通过LineSegment和Point这两个类之间的关联来反映的；学生选修课程是通过Student和Course这两个类之间的关联来表现。理解类之间如何关联很重要的原因有两个：第一，关联表明了对象之间依赖性，因为对象和系统的一部分交互，修改系统的一部分可能影响到系统的其他部分。特别是，当一个对象的接口发生变化时，很显然依赖于这个接口的客户就会受到影响。第二，类之间的关联为系统的元素和它们之间的关系提供了一个高层视图；它展示了系统的整体设计而隐藏了那些繁琐的实现细节。

类和它们的关联是用类图(class diagram)来描述的。本书中的类图遵循统一建模语言(Unified Modeling Language, UML)的标识：类表示成一个命名的方框(名字在框中间)，而关联是在两个类方框之间的一条连线。图1-4表示的是Student类和Course类之间的关联类图。关联是可以命名的并且可以有方向，在这个例子里关联命名为takes并且用一个小实心三角箭头指出了方向。它告诉我们：学生选修(takes)课程，或者说课程被学生选修。直线两端的标记数称为关联的多重性(multiplicity)，这些数字表明参与这个关联的对象的数目范围。这个类图告诉我们：学生可以选修4到6门课程，而一门课程可以被一个或更多的学生选修。

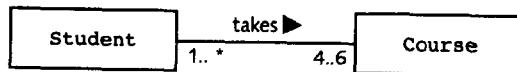


图1-4 Student和Course类之间的关联

类图中的命名类可以包括其他一些信息，包括它们的行为和其他细节等。附录C总结

了本书中所用到的UML的语法元素。

1.1.9 组合

面向对象程序设计的主要优点之一就是提供软件元素的重用。它提供了几种机制支持利用已有类来定义新类。新类获得了已有类的属性、行为和实现，这样就实现了类的重用。两种基本的重用机制是组合（composition）和继承（inheritance）。

我们已经知道：如果两个类实例之间可以交互，那么这两个类之间就存在关联。组合就是一种关联，它是一个类的实例拥有另一个类的一个或多个实例的关联。组合所表示的是两个对象之间获得一种包含（has-a）关系，即一个对象包含或拥有另一个对象。拥有其他对象的对象被称为组合体（composite），它所拥有对象被称组件（component）。这种关系在现实世界中是很普遍的：汽车里有发动机，房子包含窗户，植物包含叶子。

组合体和它的组件之间有一种特殊的关系：组合体可以直接和它的组件进行通信，并控制它们的生存期，并且只有组合体可以对它的组件进行这样的操作。组合体对象利用它的组件提供的服务来完成它的行为。举个例子来说，一个对象表示平面内的一条线段，它包含两个点对象表示它的两个端点。线段对象就是一个组合体，它包含两个组件——对应它的端点的点对象。线段对象被创建时，它的端点对象也随之生成。当要移动线段的一个端点到新的位置时，它就会向那个端点对象发送moveBy或setCoordinates消息，除了线段自己可以发送消息给它的端点对象外，没有其他任何对象可以这样做。另外，当线段的生存期结束时，它的两个端点也随之消失。

组合体对象通常包含域，这些域引用它的组件。例如，线段类可以部分定义如下：

```
public class LineSegment {

    // fields: this line segment's components
    protected Point endpoint0, endpoint1;

    // constructor
    public LineSegment(int x0, int y0, int x1, int y1) {
        endpoint0 = new Point(x0, y0);
        endpoint1 = new Point(x1, y1);
    }

    // move endpoint0 to a new location
    public void moveEndpoint0(int newX, int newY) {
        endpoint0.setCoordinates(newX, newY);
    }

    // other methods
    ...
}
```

图1-5的类图描述了LineSegment和Point类之间的关联。连接在两个类框之间的线表明它们之间的关联；位于LineSegment端的实心菱形说明这种关联是一种组合，其中LineSegment是组合体。位于Point端的数值2说明了LineSegment是由两个点组合而成的。



图1-5 线段包括两个点

有时组合体包括的组件的数目事先并不清楚，或者组件的数目在组合体的生存期中不断变化。在这种情况下，组合体就会拥有一个集合（collection），用它来保存它的组件。这个集合给组合体提供各种各样的服务，包括对组件排序、插入新组件、删除组件等等，以便于组合体对它的组件进行管理。以多边形为例，多边形是由多条直线段连接成的闭合路径。线段称为边（edge），边的端点（也就是相邻边的交点）称为顶点（vertex）。多边形允许客户进行如下操作：添加一个新顶点，删除一个顶点，按顺时针或逆时针方向对顶点进行遍历（从一个顶点转到相邻的下一个顶点）。为了支持上述操作，多边形可将它的顶点按出现顺序依次存储在一个循环链表中。循环链表提供服务，供多边形执行自身的操作时使用。例如，循环链表允许在多边形的任一位置插入新的顶点，这个服务在多边形请求插入一个新顶点时都可以使用。

1.1.10 继承

继承是软件重用的第二个基本机制。当通过继承定义一个新类时，新类获得了已存在类的域和行为。新类称为子类（child class, subclass），而已存在类称为超类（superclass）或父类（parent class）。子类又可以是其他类的父类。这就形成了继承层次结构（inheritance hierarchy），就像图1-6中的类图所描绘的样子。在这个类图中，每一个父类都与它的子类用线相连，在父类的那一边有一个空心三角形箭头指向父类。Figure类位于这个图的根；LineSegment, Point和Region是类Figure的子类；而Ellipse和Rectangle类又是Region的子类。

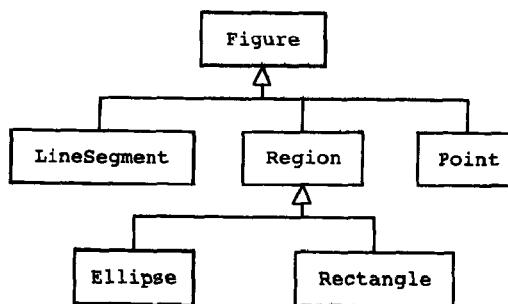


图1-6 表示继承关系的类图

每个类可以有很多个子类；除了java.lang.Object类没有父类外，每个类都有且只有一个父类。事实上，Object类位于非常庞大的继承层次结构的根，它是所有类的父类。我们所看到的类图都是那个庞大的继承层次结构的一个小部分，主要表现一小组相关类之间的关联。图1-6中并没有Object类出现，因为它与这个任务没有直接关系。

你可能已经熟悉创建一个新子类的语法。如果没有，也不用着急，请看下面Ellipse类的类定义：

```

public class Ellipse extends Region {
    ...
}
  
```

如果把extends后的内容省略掉，那么建立的新类默认为Object的子类。

简单地说，一个类对应着一个概念，它的子类对应着概念的某一特殊形式。类和它的

子类具有“是”(is-a)关系：每个子类的实例是父类的实例。参见图1-6，每个椭圆形是一个区域，而每个区域又是一个图形。

不幸的是，恰当地使用继承不像发现两个概念是否具有is-a关系那么简单，这是由于我们使用继承的主要目的是为了重用行为而不是静态属性。继承有下面三种使用方式：

- 子类定义新的属性和方法作为对它所继承的属性和方法的补充。
- 子类重新实现所继承的一个或多个方法。被子类重定义的方法称为被覆盖(overridden)了；也可以说子类覆盖(override)了特定的方法。方法覆盖后代表类的行为发生了变化：对同一个消息，子类和父类的实例调用不同的方法，产生不同的行为。
- 子类实现一个或多个其父类声明但没有实现的方法。这种情况下，这个父类就是一个抽象类，而它声明但没有实现的方法称为抽象方法。

使用继承可以创建一个类型家族。家族里的每个成员共享位于继承层次根部的父型(supertype)定义的操作，每一种类型都是父型的子型(subtype)。在图1-6中，Ellipse类型就是以下三种类型的子型：Figure、Region以及它自己(Ellipse)。图中所有的类型都支持父型Figure定义的操作，而且其中有些子型还有自己的补充操作。

一个接口A也可以作为类型家族中的父型。它的子型包括实现A的类和扩展A的接口，加上它们的子型。在图1-7中，Figure类实现Geometry接口，Geometry接口允许moveBy操作来移动一个几何图形。每个Geometry的子型都支持moveBy操作。与此相似，Region类实现了AreaGeometry接口，AreaGeometry接口声明contains操作，判断一个封闭的几何图形是否包含输入点p。这样，一个Ellipse对象就可依据客户的需要被当作一个椭圆(ellipse)、一个区域(region)、一个区域几何图形(areageometry)、一个图形(figure)或一个几何图形(geometry)。

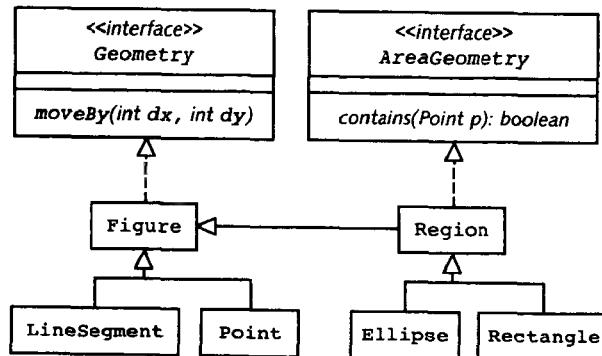


图1-7 实现接口的类图

客户可以通过一个对象的任何父型与这个对象交互。父型定义了对象的对外接口，但对象实际类型才决定它的行为，也就是说客户可以在不清楚某个对象的实际类型的情况下按父型使用它，因为它确信对象的实际类型一定会实现父型所承诺的行为。以图1-7为例，客户可以把各种不同的图形(线段、点、椭圆、矩形)都按父型Geometry处理。例如，可以定义一个过程translateAll，它的参数是geometries数组、整型dx和dy，完成的功能是对每一个几何图形沿x轴移动dx，沿y轴移动dy：

```

static void translateAll(Geometry[] geometries,
                        int dx, int dy) {
    for (int i = 0; i < geometries.length; i++) {
        Geometry geom = geometries[i];
        geom.moveBy(dx, dy);
    }
}

```

translateAll过程可以向geom发送moveBy消息，这是因为每个Geometry子型的接口中都含有moveBy。虽然并不知道每一个geom的实际类型，但它的行为是正确的，无论它是点、矩形、椭圆还是其他几何图形。多态性（polymorphism），严格说是多种形态，是指在不影响客户的情况下可以互换不同对象的能力。父型可以被看成多种形态——父型可以代表任何子型。在我们的例子中，Geometry父型代表它的任何子型。

当利用继承定义新类时，这个新子类就可以被客户使用，软件系统也因增加新的类型而扩展了。虽然有了新的类型，但软件系统还是如同以前一样工作，类的父类和使用它们的客户都不会因为子类的加入而受影响或发生变化。这就是继承非凡的一面：你可以通过子类来扩展和修改类，但类本身却不会受到任何影响。每个类都可以通过继承来修改，从这方面说它是开放的；但从客户的角度来看它们却是封闭和固定的，因为这些变化不会影响客户。继承可以在不影响类本身和其他相关元素的情况下进行类的重用。

1.1.11 设计模式与程序设计框架

无论何时遇到一个编程问题，你都会凭借自己的经验来解决它：你会考虑过去遇到的类似问题的解决方法，并利用它来解决现在的问题。可以说，你在重用一个好的解决方法。一个设计模式代表一种常见和重复出现的问题的通用解决方法。设计模式命名和描述构成解决方法的一组软件元素，并解释如何组合利用这些软件元素。它也描述解决方法适用的问题类别以及应用作出的权衡解决方法时，产生的结果。

作为设计模式的例子来看一个程序，该程序维护一个区域图形（矩形和椭圆）集合，并通过不同方式显示出它们：在一个窗口中填充输出它们，在第二窗口绘出它们的轮廓线条，在一个文本窗口中列出每一个图形的尺寸等。该问题描述如下：无论何时集合的状态发生变化——区域增加或删除，或集合中某个区域的大小和位置发生变化，窗口中的输出内容也需要跟着刷新。我们可以利用观察者设计模式（observe design pattern）来解决此类问题：区域的集合称为主体（subject），各种各样的显示窗口称为观察者（observer）。当集合的状态发生变化时，它就通知每一个显示窗口（它的观察者），然后每个窗口就相应地刷新自己。设计模式将集合和它的观察者之间的相互依赖性降为最小：集合对窗口是如何进行刷新一无所知。另外，集合可以随时登记新的观察者或注销已有的观察者，并且这对集合毫无影响。

设计模式是软件设计界成功的设计经验的结晶。与没有使用设计模式的应用相比，基于设计模式的应用更容易扩展、修改、维护和理解。设计模式还向软件开发者提供了一个共享的词汇表，利用它进行思考和交流，这样也可以促进设计模式不断壮大。

像设计模式一样，应用程序框架（application framework），简称框架（framework），也是一种设计重用方法。框架作为应用程序的主要骨架，是由一组类和接口组成，这些类和接口可以按预定的方式进行扩展和实现。框架的主要目标是在特定的应用领域里简化

程序开发。其中一个例子就是Java抽象窗口工具 (Abstract Window Toolkit, AWT)、Swing 和事件模型 (event model)，它们组合在一起形成了框架，在它们的基础上开发出具有图形用户接口 (GUI) 的应用程序。由于框架通常是和某一系统或程序设计语言联系在一起，因此它就没有设计模式那样抽象。框架的范围通常更大，它常常会包括设计模式作为其结构中的元素。

练习

1.1 当你要驾驶一辆汽车时，你会使用的抽象设备包括油门踏、刹车和方向盘，但不会用给油设备、发动机、制冷设备；但是当你的汽车坏掉时，你就要在不同的抽象中寻找问题的原因，其中包括给油设备、发动机、制冷设备和轮胎等其他设备，因为通过这些设备司机的驾驶命令被转换成动力。从司机的角度来看，一辆能正常工作的车对他来说才是有用的；而从修理工的角度看，坏掉的汽车对他来说才是有用的。对于一个复杂的事物（如汽车），从不同的角度去使用和理解，得到不同的抽象是很正常的。

由于人比汽车复杂得多，所以人可以看作是更广大范围的抽象群体。考虑不同社会角色的人的行为：父母、兄弟、儿女、朋友、服务生、银行职员、商人或医生，当你遇到某一个角色的人时，你对他将如何表现有特定的期望吗？是否有一些消息适于你发给他，而另一些则不适合？

- 1.2 一家银行在它的宣传材料中这样夸耀：“在我们的银行里，你不仅仅是一个号码。你是三个号码，跟着一个破折号，再加四个号码，另一个破折号，然后又是三个号码。”你是不是认为有些人或公司在面向对象的编程语言出现之前就已经“面向对象思维”了呢？还有哪些职业倾向于把人——业主或他们的顾客——“对象化”？他们是怎样做的呢？
- 1.3 我们为什么不能用一个循环链表来代替多边形？你能思考出一个多边形可以提供而一个链表不能提供的服务吗？
- 1.4 在图1-7的类图和我们定义的Point类的基础上，编写一个过程，它带有两个参数：一个是AreaGeometry对象数组，另一个点p。完成的功能是当每一个AreaGeometry对象都包含p时，返回一个true。这个过程的头如下所示：

```
static boolean
containsInIntersection(AreaGeometry[] g, Point p)
```

1.2 对象模型和其他程序设计模型

在本章开始，我们就强调了抽象的重要性，尤其是在处理复杂程序设计任务方面。今天的编程人员已经不再用机器语言来写程序，而是用高级语言进行开发。每一种高级语言都使用一系列的抽象，称为程序设计模型 (programming model)。程序设计模型所支持的抽象通常比计算机的底层处理的抽象要高得多，这些抽象是程序设计模型的动力。常用的程序设计模型包括：强制模型 (imperative model) (如C语言、Pascal和Fortran)，函数模型 (functional model) (Lisp, Scheme和ML)，还有定义模型 (declarative model) (Prolog)。在本书中我们学习的是面向对象的程序设计模型 (object-oriented programming)

model), 简称对象模型 (object model)。这种模型被Simula、Smalltalk、C++和Java所支持。本节余下部分将总结当今广泛应用的程序设计模型。

程序设计模型产生于20世纪50年代中期, 那时Fortran语言作为强制性的编程语言的一个例子刚刚出现。在强制程序设计模型 (imperative programming model) 下, 程序由语句或命令序列组成, 每一个语句的执行改变计算机的当前状态——通过修改内存中的内容或是产生其他影响, 如读取输入和进行输出。程序的执行会给计算机带来一系列状态变化和副作用。

强制型语言都提供赋值语句, 可以使一个名字 (即变量) 和一个值相联系。一个给定变量在不同的时间可以赋予不同的值。强制型语言通常也支持组合型存储结构, 比如数组和记录。考虑到执行流的重要性, 强制型语言通常还提供了一个控制结构的分类, 包括: 顺序结构 (例如, do this, then do that), 选择结构 (if 和switch命令), 循环 (for, do, 和while命令), 递归 (过程自己调用自己)。强制语言允许定义过程并调用这些过程。在强制模型下, 重点是描述对存储结构中的数据进行处理的过程。Fortran, PL/1, Algol, Pascal和C语言支持强制型模型。

在函数程序设计模型 (functional programming model), 也称应用设计模型 (applicative model) 下, 对表达式关注的是它的值, 而不是对它们产生的副作用。函数模型把过程看作是把输入映射到输出的抽象。函数程序设计出现于20世纪50年代后期, 以Lisp为代表, 它是LIST Programming language 首字母的缩略词。Lisp起初被主要用于人工智能。这是因为它能处理符号数据, 把程序看成是数据并能很快创建一个新程序。现在这种语言被应用于许多问题域并且由它繁衍出了很多语言, 其中最著名的有ML, Scheme, Haskell和面向对象CLOS语言。函数程序设计语言提供了相似于强制型语言控制结构, 但它却去掉了表达式的定义。例如, 函数语言提供一种其值依赖于测试表达式的值的条件表达式, 代替if语句。此外, 循环常常是利用递归实现的。函数设计语言把过程当作第一类 (first-class) 对象, 意思就是说, 过程可以作为参数传递, 也可以作为一个值返回, 还可以存储在数据结构中。在强制设计模型中, 过程和数据的区别很重要, 但在这种模型中却很模糊。

函数程序比强制程序更容易推理和进行正确性测试的原因有两个: 它们使用函数, 从数学方面很容易理解; 它们避免了副作用。然而, 标准版本的Lisp, ML和Scheme并不是纯函数的——实践要求它们提供特定的操作, 如赋值、输入和输出。

在定义模型 (declarative model), 或称基于规则的程序设计模型 (rule-based programming model) 下, 程序是由一组规则和定理组成的, 其程序提供了基础的信息 (公理) 和产生推论的方法 (规则), 但并不详细说明如何获得想要的结果。在强制设计模型程序中, 程序的逻辑是与执行语句的顺序严格相连的; 而在定义设计中, 逻辑是由它的公理和规则之间的逻辑关系决定的。这样定义程序设计逻辑和控制之间的分离, 在强制设计模型下则不能。程序员关心的不再是控制, 而是逻辑关系。这种模型下, 标准的基于规则的程序设计语言是Prolog, 用于逻辑程序设计。以规则为基础的语言缺少强制语言的控制结构。不过, 由于Prolog的控制结构会影响它的效率, 所以Prolog提供了一些控制结构用来提高其运行效率。

对象模型 (object model) 用一组相互交互的对象集合来表述所解决的问题。Java并不是第一个面向对象程序设计语言, 在它产生之前就有很多以对象模型为基础的程序设计语言。面向对象程序设计的始祖是Simula, 产生于20世纪60年代的挪威, 目的是处理仿真

问题。Simula从Algol-60借鉴了块结构和控制结构，并添加对协同程序（coroutine）的支持。（协同程序是一个过程，它的执行可以被中止，然后在它离开的原位置重新开始）。Simula-67，是Simula语言的继承者，提供了对类和对象的支持，但最终却没有广泛使用。

Smalltalk语言产生于20世纪70年代早期，由Xerox公司Palo Alto研究中心开发。它吸收过去的经验，被设计成一个纯面向对象程序设计语言。在它里面任何事物都是一个对象或一个方法，并都从一个根类里继承而来。这种语言的设计是用于鼓励面向对象的思想。在强制型语言里， $3 + 4$ 这个表达式是把3加上4而获得结果7的；而在Smalltalk里，表达式 $3 + 4$ 被视为一个消息：对象3是消息接收者，+是消息名，对象4作为参数。为了得出 $3 + 4$ 的值，你向对象3发送消息+4，这样得到结果对象7。这是一个二元消息表达式的例子。Smalltalk还定义了另外两种形式的消息表达式。一元消息是一个不带参数的消息。例如，表达式4 factorial（阶乘）是指给对象4发送消息factorial，这样获得对象24。一个关键字消息指它的参数被定义为关键字的消息。例如，表达式anArray at: 2 put: 5是指把对象5放在数组anArray的下标为2的位置上。这个消息是用anArray的名为at:put:的方法处理的，关键字at:和put:把2和5匹配为方法的参数。

Smalltalk的设计意图简单，不仅适合专业编程人员使用，也适用于非专业人士和孩子们。发送消息和定义方法的语法都很简单，而且还避免了其他语言中大量的复杂操作（如对于操作者的优先权和关联问题，或是在强制性语言里大量复杂的控制结构）。就像在Java里，对象在堆中分配并通过引用使用，而且当对象不再被引用时，自动被释放。Smalltalk是作为一个集成开发环境出现的，包括编辑器窗口、图形窗口、查看继承层次结构的浏览器窗口、解释器，以及其他特性。

源自于C语言的C++语言产生于20世纪80年代，由贝尔试验室的Bjarne Stroustrup开发。从C到C++的第一个语言称为C with Classes，它吸取Simula面向对象的一些特点，但用的是C的语法。从C with Classes到C++，添加了动态方法绑定（虚函数）、函数和操作符重载，以及引用类型这些机制；这些都是在1985年发布C++ 1.0版时首次出现的。从1985年到1990年，C++不断增加大量的用户输入的处理。在1990年发布的C++ 3.0版包含有模板和异常处理。

在最近几年，C++成为一种非常受欢迎的语言，这是由于它与C语言兼容，C程序可以作为C++程序进行编译，这样可以使C程序员按照他们自己的方式学习和吸收C++的特性。另外，两门语言在语法和语义上的相似，可以使对C语言已经有了一定了解的人们更加容易地学习C++。C++拥有一些其他面向对象语言没有的特点。模板提供了一种方法，利用它可以定义一般类型的类作为一个类族的模板；例如，你可以定义一个集合模板，利用它可以生成各种存储特定类型的集合类（用来存储整数的集合类，用来存储点的集合类）。操作符重载允许你按自己的方式定义像+、*和<这样的操作符。多重继承允许一个新类从任何数目的其他类（即它的父类）继承实现和接口（对比而言，Java只支持接口的多重继承而不支持实现的继承）。C++拥有丰富的功能强大的特点，但另一面，作为一种复杂语言要正确使用存在一定难度。

Java产生于1991年，当时Sun Microsystems公司的一组工程师在James Gosling的带领下开发一种用于有线电视转接盒、盒式磁带录相机、微波炉等消费类电器的编程语言。到1994年，工程师们发现他们开发的语言可以很好地应用于因特网上——它是面向对象的、安全的、结构中立的、解释性的、多线程的语言。他们用Java开发了一个称为HotJava的

网络浏览器，在1995年第一次当众演示。当1995年后期被Sun正式发行时，Java主要用于创建小应用程序（applet），小应用程序可以被嵌入在网页中并由浏览器来执行处理。在随后的几年中，Java已经成为头等开发语言，被广泛应用在各种各样的系统开发中，从嵌入式设备到大型应用程序。最新版本的Java 2于1999年发行，新特性中包含用于二维图形设计的Java 2D，也就是本书所要使用的功能。

Java被广泛采用的原因之一是它的平台独立性，也就是“一次性编写，跨平台运行”。当Java源代码被编译完后，它产生的可执行代码是一组Java字节码（Java bytecode）指令，这就是被Java解释器——Java虚拟机（Java virtual machine，JVM）加工处理的语言。Java可执行代码可以在任何一台安装JVM的机器上运行，由于所有流行的操作系统都有JVM，所以Java编译程序可以在因特网上发布，下载后可以在所有的平台上运行，这使Java成为通用的因特网语言。另外由于Java包含一系列标准类，并且Java程序所需要的绝大多数类都已经在执行这些程序的机器上，所以为了共享一个Java程序，没有必要传输组成这个程序的所有类，只需传输那些对程序特别的类。一个小的Java程序可以封装强大的功能。

第2章 过程抽象

过程之所以重要有两个原因：首先，方法是与对象相关的过程，如果没有理解过程，就不能充分理解对象的概念；其次，许多不同类型的问题在强制程序设计模型下可以得到很好的解决，在这种模型下，过程是按照一定的预定顺序执行指令和调用其他过程来进行运算的。

在本章中，我们将要学习一种所有支持过程创建和使用的语言共同使用的抽象形式。过程抽象是把过程作为一个抽象操作。这种抽象隐藏了过程的具体实现。它主要关注于过程的功能，而不是过程如何实现这些功能。

在2.1节中，我们要讨论过程抽象的基本优点，它允许将过程看作一种抽象操作。2.2节介绍表达抽象操作的符号。2.3节讲述Java的异常机制（过程可能抛出的异常是过程说明的一部分）。本章余下各节介绍可以通过过程抽象实现的两种重要方法：过程分解（程序开发规程）（2.4节）和递归编程技巧（2.5节）。

2.1 抽象操作和过程

一个过程可以从两个方面来看。首先，过程代表了一种抽象操作（abstract operation），获得输入、产生输出和副作用；其次，过程描述了一种计算过程（computational process），这是一些步骤，通过它们实现操作要完成的功能。过程的这两个方面紧密联系：计算过程实现了抽象操作。

过程的定义同时体现了这两个方面。例如，考虑下面过程square的定义，它用于求一个正整数的平方：

```
static int square(int k) {
    return k * k;
}
```

一方面，这个过程定义描述了一个抽象操作。这个操作需要输入一个正整数，并返回所输入值的平方。另一方面，过程定义描述了执行这个过程的操作步骤：输入K，K乘以它自身并返回结果。总之，过程定义既描述了过程所要做的工作（要做的抽象操作）也描述了它是如何完成这些工作的（实现方式）。

过程抽象（procedural abstraction）将一个过程看作隐藏了它的运算过程的抽象操作。从调用过程的客户角度来看，这是一种非常恰当的描述。对客户而言，过程所实现功能的正确性是至关重要的，而对于它如何实现客户不必关心。举一个简单的例子。勾股定理是众所周知的：在一个直角三角形中，两条直角边的长度a和b，斜边的长度c可以利用公式 $c = \sqrt{a^2 + b^2}$ 求得。以下的程序就是利用这个公式来求两个边长为a和b的直角三角形的斜边长c：

```
static double hypotenuse(int a, int b) {
    int c2 = square(a) + square(b);
    return Math.sqrt(c2);    // returns the square root of c2
}
```

过程hypotenuse的实现依赖于过程square提供的操作，但它不依赖于square是如何实现的。过程square可以按下面的方式来实现：

```
// version 1
static int square(int k) {
    return k * k;
}
```

另外，这个过程可以用一种效率较低的方式来实现：

```
// version 2
static int square(int k) {
    int res = 0;
    for (int i = 0; i < k; i++)
        res += k;
    return res;
}
```

同样，过程square也可以按如下方式来实现：

```
// version 3
static int square(int k) {
    float s = (float) Math.exp(2.0 * Math.log(k));
    return Math.round(s);
}
```

三个版本的square代表了相同的抽象操作：当传递一个正整数调用过程时，它们都返回输入值的平方。另外，不同square的实现版本，不会影响任何一个调用square的过程的正确性。无论使用哪一个square版本，过程hypotenuse都表示相同的抽象操作。

过程抽象有两个主要优点。首先，通过将过程看作抽象操作的方式，程序员可以在无需知道过程是如何实现的情况下使用它们。过程可以在任何时间或者是由任何人编写或者它们属于一个库。在过程hypotenuse的例子中，程序员可以在不知道过程square如何实现的情况下编写hypotenuse过程。因为过程square提供了很简单的实现，这个例子可能看起来很简单。但是过程hypotenuse也调用了求平方根的过程java.lang.Math.sqrt，这也说明过程抽象非常方便，使我们无需编写求平方根的过程，也无须知道Java版本的求平方根是如何工作的。

过程抽象的第二个优点在于，只要抽象操作功能是确定的，即使过程的实现被修改，也不会影响使用这个过程的程序。调用过程的客户依赖于过程所要实现的操作，而非它的具体实现方式。例如，过程square在过程hypotenuse中的使用，如果用过程square版本2替换版本1，过程hypotenuse仍然正确，因为过程square的这两个版本表示了相同的抽象操作，尽管它们的实现方式不同。

前面已经说过，在过程抽象中，过程被看作一种抽象操作，是一种获得输入、产生输出和／或副作用的“黑盒子”。那么，描述一个过程所代表的抽象操作的最好方式是什么呢？

一种方式是提供过程的完整定义。毕竟，过程的实现方式揭示了过程所要实现的功能。但是，实现方式可能难以理解——它可能会错综复杂，还可能要依赖于其他的过程和对象，使得我们必须刨根问底地理解它们。另外，过程定义有时是不可行的。因为在任何情况下，过程抽象的一个优点就是隐藏了过程的实现方式，所以人们希望有一种方式能够描述抽象操作，而隐藏它的具体实现方式。

另一种描述抽象操作的较好的方式是描述它所需的输入值范围和它所产生的输出结果

和额外作用。以下是对过程square的一种描述：

```
static int square(int k)
  // input: a positive integer k
  // output: an integer denoting k squared
```

操作的结果可能有其他副作用，如读取一个流或写入一个流，或者修改某个对象的状态。下面例子中过程的副作用是把计算结果写入标准输出流：

```
static void printSquare(int k) {
  // input: a positive integer k
  // side effect: prints k squared
  // to the standard output stream
  System.out.print(square(k));
}
```

下面这个过程的副作用是修改调用时传递给过程的对象的状态。过程translate1要求输入一个点p和两个整型值dx和dy，将p的坐标在x轴上移动dx个单位，在y轴上移动dy个单位：

```
static void translate1(Point p, int dx, int dy) {
  // input: a nonnull point p, and two integers dx and dy
  // side effect: translates point p by dx and dy
  p.setX(p.getX() + dx);
  p.setY(p.getY() + dy);
}
```

过程translate1的副作用是改变点p的坐标，但没有返回值。例如，

```
Point p = new Point(2, 3);
translate1(p, 4, 5);
System.out.println("p: " + p); // p: (6,8)
```

明确返回值和副作用的区别是很重要的。这是两种不同的方式，过程的操作通过它们和程序其他部分交换信息。对于返回值，使用代码从它调用的过程中接收一个返回值，并常常把它赋给一个变量；而对于副作用，过程的执行经常改变一个或多个对象的状态。与过程translate1相比，过程translate2仍要求输入参数点p、整型dx和dy，并返回一个新点，新点是点p加dx和dy后的点。点p在过程操作中不变：

```
static Point translate2(Point p, int dx, int dy) {
  // input: a nonnull point p, and two integers dx and dy
  // output: returns a new point equal
  // to (p.getX() + dx, p.getY() + dy)
  int x = p.getX() + dx;
  int y = p.getY() + dy;
  Point q = new Point(x, y);
  return q;
}
```

过程translate2返回一个值，但不产生其他副作用。尽管它创建并返回了一个新的点对象，但它并没有修改任何已存在的对象的状态；尤其是并没有改变点p的状态。与此相反，过程translate1说明改变了p点状态，但没有返回值。以下的代码段显示了这两个过程的区别：

```
Point p = new Point(2, 3);
Point q = translate2(p, 1, 2);
System.out.println("p: " + p); // p: (2,3)
System.out.println("q: " + q); // q: (3,5)
translate1(p, 4, 5);
System.out.println("p: " + p); // p: (6,8)
```

练习

2.1 对于一个过程来说，既有返回值又产生副作用当然是可以的，请实现以下过程：

```
static Point translate3(Point p,
                        int dx, int dy)
// input: a nonnull point p, and
//   integers dx and dy
// side effect: translates p by dx and dy
// output: point p after it is translated
```

2.2 将下面的两个方法添加到第1章的Point类中。第一个方法按它的两个输入dx和dy移动点，第二个方法是计算这个点和输入点p之间的距离：

```
// methods of Point class
public void translate(int dx, int dy)
    // side effect: translates this point by dx and dy

public double distance(Point p)
    // input: a nonnull point p
    // output: distance between this point and point p
```

可用下面的短程序来测试你的方法：

```
public class TryNewPointMethods {
    public static void main(String[] args) {
        Point p = new Point();
        Point q = new Point(p);
        q.translate(3, 4);
        System.out.println("q: " + q);
        System.out.println("dist: " + p.distance(q));
    }
}
```

2.2 过程说明

过程说明描述了过程的行为，且独立于它的实现方式。这种说明详细指出过程与其客户之间的协议。在过程square的例子

```
static int square(int k)
// input: a positive integer k
// output: an integer denoting k squared
```

中，客户确保使用一个正整型的值来调用square。相应地，过程square确保返回一个整型值，该值等于输入值的平方。

这个协议由过程的前置条件（precondition）和后置条件（postcondition）所决定：前置条件是任何调用过程的客户所必须满足的条件；后置条件是过程在客户满足了前置条件的前提下确保会实现的条件。

为了说明过程的前置条件和后置条件，我们给出了一个紧随过程头的说明注释。注释包含以下三个子句：

- 需求子句（requires clause）说明过程的前置条件，即使用代码时必须满足的条件。
- 修改子句（modifies clause）列出了被过程改变的对象名。这些名字经常是作为过程的输入，但它们通常会包含过程可能改变其状态的所有元素。
- 作用子句（effects clause）描述过程作用在输入上的行为，这些输入必须是没有被前置条件排除的。这一子句将过程的后置条件和它的合法输入联系在一起，但是并没

有说明当需求子句没被满足时过程的行为。

当前置条件永远都为真时（即用来调用过程的任何输入都是合法的），需求子句可以被省略。当过程不产生副作用的时候，修改子句可以省略。因为每一个有用的过程都会产生一些改变，所以作用子句通常是被说明的。下面来看几个使用这一表示法的例子。

在过程square的例子中，前置条件是过程必须使用一个正整型值来调用，后置条件是过程返回一个等于输入值平方的整型值。虽然在过程头中声明了输入值和返回值的类型，但是注释还至少应该传达过程前置条件和后置条件的其他方面：

```
static int square(int k)
  // REQUIRES: k is positive.
  // EFFECTS: Returns k squared.
```

另外一个例子是上一节中最后的过程translate1定义：

```
static void translate1(Point p, int dx, int dy) {
  // REQUIRES: p is not null.
  // MODIFIES: p
  // EFFECTS: Translates p; that is,
  //   p_post==(p.getX() + dx, p.getY() + dy).
  p.setX(p.getX() + dx);
  p.setY(p.getY() + dy);
}
```

在作用子句中，`p_post`是过程返回时点p的状态，`p`是当过程被调用时自己的状态。

以下重复了过程translate2的定义并且包含它的说明注释：

```
static Point translate2(Point p, int dx, int dy) {
  // REQUIRES: p is not null.
  // EFFECTS: Returns a new point q
  //   where q==(p.getX() + dx, p.getY() + dy).
  int x = p.getX() + dx;
  int y = p.getY() + dy;
  Point q = new Point(x, y);
  return q;
}
```

过程translate2没有修改子句，因为它没有改变点p和其他任何对象的状态。

过程translate3与translate1类似，但是它还返回改变后的点：

```
static Point translate3(Point p, int dx, int dy) {
  // REQUIRES: p is not null.
  // MODIFIES: p
  // EFFECTS: Returns p_post where p_post is
  //   p translated by dx and dy:
  //   p_post == (p.getX() + dx, p.getY() + dy).
  p.setX(p.getX() + dx);
  p.setY(p.getY() + dy);
  return p;
}
```

过程translate1, translate2, translate3的说明尽管很相似，但它们显示了不同的行为。

在下一章中，我们可以看到过程抽象在对象模型中发挥着重要的作用。客户可以通过向对象发送一个消息来调用一个过程。客户将过程当做抽象操作来使用，这种抽象操作可以被看作输入对输出和副作用的映射，并由过程的需求子句、修改子句和作用子句来描述，而客户并不知道这个操作是如何实现的。事实上严格地说，客户调用对象的过程这种说法是不恰当的。当客户向一个对象发送消息的时候，一般来说，它是不知道哪一

个方法会被调用的。

使用过程抽象，对象的客户视对象提供的过程为抽象操作。为一个方法注释需求子句、修改子句、作用子句，这个方法就被描述为抽象操作。在本书中，我们将使用这种说明方法，包括在下一章中类方法的说明。

形成断言

过程表达过程和它的调用者之间的一项协议 (contract)。这个协议的条款是由前置条件和后置条件共同说明的。由需求子句说明的前置条件表述调用者的义务。后置条件描述当调用者提供了所有合法输入时过程的义务。作用子句描述了这些义务为所有合法的输入的函数，后置条件经常还包括副作用，修改子句列出状态可能被副作用改变的所有对象。

由于从协议观点看过程调用描述了职责，所以当出现错误的时候，我们可以辨别出错误的来源。考虑一下当客户调用过程时会发生什么事情：如果客户违反了任何一条前置条件，就会出错；当客户满足了所有的前置条件，但是过程的后置条件没有得到实现，过程仍然会出错。例如，当客户以

```
square(-2);
```

调用过程 `square` 时出现了错误，因为违反了过程的前置条件，传递了一个负数。而当调用

```
square(4);
```

时返回了值 7，过程 `square` 在满足了前置条件的情况下出现了错误，它没有满足后置条件。

这种观察导致形成断言的调试方法，凭借断言一个过程的前置条件和后置条件可以当作执行的一部分来检查。（更一般地说一个断言 (assertion) 是用来检查满足特定条件的表达式。）当断言条件成立时，执行中的程序断定某种条件应该成立，而如果条件不成立就会出现错误。以下是带有前置条件和后置条件断言的过程 `translate2`：

```
static Point translate2(Point p, int dx, int dy) {
    // REQUIRES: p is not null.
    // EFFECTS: Returns a new point q,
    //           where q==(p.getX() + dx, p.getY() + dy).
    Assert.pre(p != null, "argument p is null");
    int x = p.getX() + dx;
    int y = p.getY() + dy;
    Point q = new Point(x, y);
    Assert.post(q.getX() == p.getX() + dx,
                "result not translated by dx");
    Assert.post(q.getY() == p.getY() + dy,
                "result not translated by dy");
    return q;
}
```

如果用 `null` 作为第一个参数调用过程 `translate2`，就违反了需求子句，则方法 `Assert.pre` 会抛出一个新的异常对象（异常将会在下一节中讲到）。异常对象包含了错误消息，是一个描述错误的字符串。如果抛出的异常没有被捕获到，Java 解释器会打印出异常的出错消息和调用堆栈的记录，然后退出程序。例如，语句

```
translate2(null, 2, 3);
```

会在控制窗口中产生以下消息：

```
> java TryAssertions
Exception in thread "main" banana.FailedConditionException:
  precondition failed: argument p is not null
    at banana.Assert.pre(Assert.java:9)
    at TryAssertions.translate2(TryAssertions.java:12)
    at TryAssertions.main(TryAssertions.java:8)
```

前置条件失败的事实告诉我们是客户的错误。

另外，假设translate2的实现出现了错误。例如，假定声明和初始化变量y语句被错误地编码如下：

```
int y = p.getY() * dy;
```

那么，对合法指令

```
translate2(new Point(2, 3), 4, 5);
```

的响应，解释器会产生出错信息如下：

```
postcondition failed: result not translated by dy
```

因为后置条件失败，我们可以知道是过程translate2的错误。

Assert类中定义了静态方法pre和post，用它们分别断言前置条件和后置条件。方法pre用两个参数调用：一个布尔表达式代表前置条件，一个是字符串消息msg。如果布尔表达式的值为真，那么方法不做任何事情；而当其为假时，说明前置条件失败，方法抛出FailedConditionException类的一个实例。这个异常对象包含描述出错信息的字符串。为了检查后置条件，方法Assert.post使用相似的定义。Assert类也定义了一个方法condition来检查一般的条件。以下为类定义：

```
public class Assert {
    public static void pre(boolean test, String msg)
        throws FailedConditionException {
        // REQUIRES: msg is not null.
        // EFFECTS: If test is false
        //   throws FailedConditionException
        //   indicating the failed precondition.
        if (!test) {
            String s = "precondition failed: " + msg;
            throw new FailedConditionException(s);
        }
    }

    public static void post(boolean test, String msg)
        throws FailedConditionException {
        // REQUIRES: msg is not null.
        // EFFECTS: If test is false
        //   throws FailedConditionException
        //   indicating the failed postcondition.
        if (!test) {
            String s = "postcondition failed: " + msg;
            throw new FailedConditionException(s);
        }
    }

    public static void condition(boolean test, String msg)
        throws FailedConditionException {
        // EFFECTS: If test is false
```

```
//     throws FailedConditionException
//     indicating the failed condition.
if (!test) {
    String s = "condition failed: " + msg;
    throw new FailedConditionException(s);
}
```

以下是FailedConditionException类的定义：

```
public class FailedConditionException
    extends RuntimeException {
    public FailedConditionException(String msg) {
        super(msg); }
    public FailedConditionException() { }
}
```

形成断言对调试很有用。尽管断言在调试后可以被去掉，但是断言的使用增加了许多代码行，使程序执行效率降低。因为正式的断言解释过程的行为不如英文注释和伪代码清楚，所以在本书的代码中没有使用断言。但是你可以使用**Assert**类来调试自己的程序。

2.3 异常

在过程体中包含断言是检查错误的一般方法。当一个断言条件失败时，被抛出的对象类型总是相同的——`FailedConditionException`类的一个实例。尽管与`FailConditionException`类相关的字符串会告诉我们一些出错信息（如它是否涉及前置条件或后置条件），但是异常对象的类型并没有说明具体的错误。此外，可能会有关于失败条件的信息，但是`FailedConditionException`类没有提供存储这些信息的域。

Java提供了更为通用的异常处理机制 (exception mechanism) 来处理错误和异常。异常是一种不同寻常的情况，但是在某些情况下却会发生。异常的例子包括打开一个不存在的文件、读文件时越界、用零做除数、间接引用一个空引用。Java的异常机制允许代码同时检测和处理异常，无需聚集在不出现任何异常时产生的“正常”处理代码。

异常处理通常由抛出异常和捕捉异常完成。当有不正常情况出现时，就会有异常被抛出。事实上被抛出的通常是封装了描述异常信息的对象。这个对象的类型是由异常的性质决定的。抛出异常有两种方法：在过程中使用**throw**语句，或是进行非法的底层操作时由Java解释器抛出。

当过程或是操作抛出一个异常时，调用它的代码可能会捕捉异常并进行处理。另外，代码允许异常传播到过程调用栈，使其他的调用过程有获得处理这个异常的机会。如果异常最终没有被捕获，Java解释器会捕捉它，并终止过程的执行且打印出有用的错误信息。

抛出异常是过程说明的一部分，同返回一个值或是产生一个副作用很相像。正如过程的返回值类型被过程头声明并且被作用子句解释，过程抛出的异常类型也须用此种方式声明和解释。过程抛出的异常在两个地方指明：在过程头的throws子句中，在说明注释的作用子句中。例如，下面是变换点的过程的说明：

```
static void translate4(Point p, int dx, int dy)
    throws NullPointerException
// MODIFIES: p
// EFFECTS: If p is null throws NullPointerException;
```

```
// else translates p by dx and dy:  
// p_post===(p.getX()+dx,p.getY()+dy).
```

当过程translate4的第一个参数为空时，它会抛出NullPointerException异常。重要的是在它的说明中表达了这种行为。translate4与上一节中的translate1相似，都是通过输入的dx和dy值来改变输入点p。它们的不同之处在于translate1中的前置条件不允许p为空，而translate4的第一个参数为空是合法的。当发生这种情况时，translate4会抛出一个NullPointerException类型的异常对象。通过捕捉这个异常，调用过程的代码可以探测到发生的情况并且做出相应的反应。

当过程抛出多种类型的异常时，过程的作用子句应该描述出各种异常类型发生的条件，且过程头的throws子句也应该列出过程的所有异常类型。以下是找出整数数组中最小整数的位置的过程的说明：

```
static int min(int[] a)  
    throws ZeroArraySizeException,  
           NullPointerException  
// EFFECTS: If a is null throws NullPointerException;  
//   else if a has size zero throws  
//     ZeroArraySizeException; else returns the index  
//     of some smallest item in a.
```

2.3.1 受检查异常和不受检查异常

NullPointerException是Java预定义的许多异常类中一个，所有的类都属于图2-1的类层次结构。这个层次结构的根是java.lang.Throwable类，它是所有可抛出错误和异常对象的父型。Error类的子类包含涉及Java解释器的错误或者资源不足的错误，你的程序永远无需抛出这些类型的对象。在Exception类的子型中，Java会区分受检查异常（checked exception）和不受检查异常（unchecked exception）。特别地，RuntimeException类的子型都是不受检查异常，而Exception类的其他所有子型都是受检查异常。

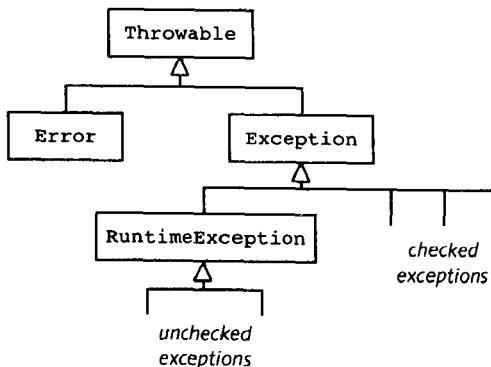


图2-1 RuntimeException的子型是不受检查异常而其他所有异常都是受检查异常

不受检查异常一般来自程序设计错误，包括算术异常，例如除数为零（ArithmaticException）、访问数组越界（ArrayIndexOutOfBoundsException）、数字格式错误异常（NumberFormatException）、当要求一个有效对象引用时使用null异常（NullPointerException）。可以扩展RuntimeException类来定义一个新的不受检查

异常类。回忆我们是如何定义类**FailedConditionException**的，它的实例是由**Assert**类的静态方法抛出（2.2.1节）：

```
public class FailedConditionException
    extends RuntimeException {
    public FailedConditionException (String msg) {
        super(msg); }
    public FailedConditionException () { }
}
```

受检查异常代表了由于环境因素而程序自身无法控制的情况。但是程序可以智能地响应这些情况。例如读文件时越界（**EOFException**）、试图打开一个不存在的文件（**FileNotFoundException**）、试图打开一个错误格式的URL（**MalformedURLException**）等。为了定义一个新的受检查异常类，你必须扩展**java.lang.Exception**类：

```
public class MyCheckedException extends Exception {
    ...
}
```

受检查异常是接受编译器检查的，这也是它们被如此命名的原因。使用受检查异常须遵守以下两条规则：

- 如果一个过程可以抛出一个受检查异常，这个异常的类型（或是子型）必须在过程头的**throws**子句中列出。换句话说，过程必须声明所有它可能抛出的受检查异常。
- 如果代码调用了一个可以抛出受检查异常的过程，那么代码段必须处理这些异常：捕捉这个异常，或者它自己也会抛出这个类型的异常。

如果没有遵守这两条规则，编译器会报告出现错误。而不受检查异常不必遵守这两条规则：过程可以随意选择是否声明抛出的不受检查异常，代码可以随意选择是否处理它产生的不受检查异常。

我的策略是声明过程定义中出现的每一个异常——无论是受检查异常还是不受检查异常。换句话说，在过程的作用子句中提到过的异常，在过程的**throws**子句中都要声明。

2.3.2 抛出异常

要抛出一个异常，需要使用**throw**语句，它的参数是某个**Exception**类的子型的实例。例如，

```
throw new FileNotFoundException();
```

异常创建时通常会伴随有一个错误消息——描述异常的字符串

```
throw new FileNotFoundException("cannot find file "+file);
```

其中**file**是指文件名。为了获得异常对象的错误信息，你需要对它发送**getMessage**消息。

Assert类的静态方法就是抛出异常的过程的例子。**translate4**过程的下述实现是另一个例子：

```
static void translate4(Point p, int dx, int dy)
    throws NullPointerException {
    // MODIFIES: p
    // EFFECTS: If p is null throws NullPointerException;
    //   else translates p;
    //   p_post==(p.getX()+dx, p.getY()+dy).
    if (p == null) throw new NullPointerException();
```

```

    p.setX(p.getX() + dx);
    p.setY(p.getY() + dy);
}

```

2.3.3 捕捉异常

我们使用try块和catch块来捕捉和处理异常。导致异常的代码放在try块中。处理各种可能出现的异常类型的代码包含在catch块中。例如，下列代码段试图将字符串s解析为整型值：

```

// string s is expected to denote a signed decimal integer
try {
    int i = Integer.parseInt(s);
} catch (NumberFormatException e) {
    // handle number format exception here
    ...
}

```

如果字符串s形式错误，过程Integer.parseInt的调用会抛出一个NumberFormatException类型的异常。抛出异常对象会被catch捕获，并绑定在catch块的参数e中，然后执行这个catch块。在catch块中，抛出的异常可以通过参数e引用。

当有多个catch块处理同一个try时，所有的catch块都会被从上到下地检查，直到找到第一个参数可以接受这个异常的catch块被执行，即如果catch块的参数是异常类型的同类型或是父型时，它就执行。下面是一个涉及两个catch块的过程，使用了在2.3节开始时指定的min过程：

```

// a is an array of ints
try {
    int indx = min(a);
} catch (NullPointerException e) {
    // handle case where a is null
    ...
} catch (ZeroArraySizeException e) {
    // handle case where a has length zero
    ...
}

```

只有第一个可以接受异常的catch子句执行，后面的catch子句不能执行，尽管它们也可以接收异常。如果没有catch子句可以接收抛出的异常，异常被向上传播到调用方法中，检查是否有相匹配的catch子句。异常通常会一级一级在调用堆栈(call stack)中向上传播。方法激活的顺序是从当前方法到最外层的第一个方法。每个方法都有捕获异常的机会，直到有一个方法可以捕捉这个异常。如果没有方法捕捉这个异常，那么它被传递到Java虚拟机。

2.3.4 处理异常

当发生异常时，调用过程可以用两种方式处理异常。第一种是允许异常向上传播。在这种情况下，异常必须被声明为调用过程自身可能抛出的异常类型（至少在受检查异常的情况下应如此）。以下过程与过程translate4有相同的说明，但是实现方式略有不同：translate4中的throw语句在下列实现中被省略：

```
static void translate5(Point p, int dx, int dy)
```

```

    throws NullPointerException {
// MODIFIES: p
// EFFECTS: If p is null throws NullPointerException;
// else translates p:
//   p_post==(p.getX()+dx, p.getY()+dy).
p.setX(p.getX() + dx);
p.setY(p.getY() + dy);
}

```

当过程translate5的第一个参数值为null时，第一条指令使得值为null的变量p被非法间接引用。相应地，Java解释器会抛出NullPointerException异常。因为translate5并未捕获这个异常，该异常被传播到它的调用者。这里请注意translate5在它的throws子句中声明过并且在作用子句中解释过NullPointerException异常。

第二种处理异常的方法是使用Java的try和catch块机制来捕捉它。在此种情况下，可能抛出异常的代码在try块中，处理异常的代码被放在随后的catch块中。处理异常的代码可以有两种相应的方式：第一，当进行某些可能的操作后，它抛出了一个自己的异常对象，这个异常对象可能与它捕捉的异常是同一种类型的，或者是它自己创建的新的异常对象；第二，异常处理代码可以解决所有问题，而没有必要再抛出一个异常。

以下是第一种情况的示例，即过程捕捉一个异常并且通过再抛出一个自己的异常来作为响应。这个过程改变了点数组里的每一个点：

```

static void translatePoints1(Point[] points, int dx, int dy)
throws NullPointerException, IllegalArgumentException {
// MODIFIES: points
// EFFECTS: If points is null throws
//   NullPointerException; else if
//   points[i] is null for some i throws
//   IllegalArgumentException; else translates each
//   points[i] by dx and dy.
if (points == null) throw new NullPointerException();
try {
    for (int i = 0; i < points.length; i++)
        translate5(points[i], dx, dy);
} catch (NullPointerException e) {
    throw new IllegalArgumentException();
}
}

```

try块中出现重复调用translate5的for循环。当用null调用translate5时，它会抛出NullPointerException异常。这样会使执行退出try块，进入catch块。这样参数e被绑定到抛出的异常对象，并创建一个新的IllegalArgumentException类型异常并抛出它。

过程处理异常的第二种方法是自己采取措施。下面是这样一个例子，功能同上面例子相同，但与前面不同的是，如果points[i]为null时，置它们为新点(dx, dy)。下面是该过程：

```

static void translatePoints2(Point[] points, int dx, int dy)
throws NullPointerException {
// MODIFIES: points
// EFFECTS: If points is null throws
//   NullPointerException; else for
//   each i, if points[i] is null sets
//   points[i] to new Point(dx,dy), else translates
//   points[i] by dx and dy.
if (points == null) throw new NullPointerException();

```

```

for (int i = 0; i < points.length; i++) {
    try {
        translate5(points[i], dx, dy);
    } catch (NullPointerException e) {
        points[i] = new Point(dx, dy);
    }
}
}

```

2.3.5 使用异常

除了抛出异常和捕捉异常，代码还可以用另外两种方法确定和处理问题。第一，过程可以返回一个特殊值表示发生过异常，并且使用不同的特殊值来表示不同的异常类型。第二，过程可以设置一个对客户来说是可见的标志。事实上，这两种方法已经使用了几十年，并且在缺少明确的异常机制的编程语言中是十分必要的。然而，Java的异常机制有很多优点，至少可以使过程的正常处理与异常处理分离。在try块中封装了正常处理的代码，在随后的catch块中包含了异常处理代码。

在使用异常进行错误检查方面有两种思想学派。防错编程技术建议方法要检查它们是否被正确使用。当一个方法被调用时，它会监测需求子句是否满足，如果没有满足就抛出一个适当的异常。这种方法在调试时很有用，它可以检测出错误发生处。它也可以为错误使用方法的客户提供一张安全网。

第二种思想学派建议严格按照说明编程。根据这种观点，方法只能按照它的说明来抛出异常，如果违反了它的需求子句，并不用采取任何特别的行动。这就直接将满足需求子句的重担交给了客户。关于这种方法有以下几点争论：第一，错误检查需要花费时间，在客户确保前置条件的情况下（当程序是正确的前提下），额外时间就被浪费了；第二，防错编程会导致客户程序设计者的惰性，他们会认为无论什么时候错误地使用了过程，过程都会捕捉这个问题。也许更重要的是，方法检测到它的前置条件没有满足时抛出异常，这就违反了方法说明的精神。抛出异常是过程的需求子句得到满足时过程发生动作的一部分。在调试中，最好能够包含检测前置条件的断言，但是只有当前置条件得到满足时，过程才能从根本上确保必然发生的行为。

这就说明设计过程时，它的需求子句是空的，将客户的负担减少到最少，将过程执行的错误检查最大化。相反，严格的前置条件经常是有用的。例如，过程的设计在确保前置条件的环境下使用。同样经常发生的情况是，在满足某些特定前置条件时，有可能改进过程的性能。关键是异常抛出是过程确保发生行为的一部分，并且只有在过程的需求子句得到满足时，这些行为才有意义。

练习

2.3 给出下列Java过程的说明注释——需求子句、修改子句、作用子句。下列方法所在的类除了`java.util.Arrays`类之外都属于`java.lang`包。在实例方法（没有被声明为`static`类型）中，你的注释要包括术语`this`，它指当前调用方法所属的对象（被声明为`static`类型的类方法，没有当前对象）。

- (a) `static double Math.random()`
- (b) `static void System.gc()`

```
(c) void String.concat(String str)
(d) void String String.charAt(int indx)
(e) String String.replace(char oldChar, char newChar)
(f) int Integer.parseInt(String s)
(g) static void Arrays.sort(int[] a, int fromIndex,
                           int toIndex)
```

2.4 实现下面的过程：

```
static int min(int[] a, int lo, int hi)
    throws NullPointerException,
    ZeroArraySizeException, IllegalArgumentException
// EFFECTS: If a is null throws NullPointerException;
// else if a is empty throws ZeroArraySizeException;
// else if 0 <= lo <= hi < a.length returns the index
// of some smallest element in the subarray of a from
// index lo through index hi inclusive;
// else throws IllegalArgumentException.
```

2.5 根据上一个练习中的三个参数的过程min实现下面的过程：

```
static int min(int[] a)
throws ZeroArraySizeException, NullPointerException
// EFFECTS: If a is null throws NullPointerException;
// else if a is empty throws ZeroArraySizeException;
// else returns index of some smallest item in a.
```

2.6 用返回特殊值来指明是否发生过异常，这种方法有什么缺点？设置一个标志变量来指明异常，这种方法有什么缺点呢？比起Java的异常机制，这两种方法有哪些优点呢？ ---

2.4 过程分解

过程分解 (procedure decomposition)，又称为自顶向下结构化设计 (top-down structured design) 和逐步细化 (stepwise refinement)，是结构化程序设计中经过长时间考验的规范。它是结构化编程语言如C、Pascal中最重要的设计方法。因为面向对象化语言如Java也要依赖过程，所以过程分解在面向对象编程中也是非常重要的。

假如给定一个有待解决的问题，这种思想是将问题分解为多个子问题，并且逐个解决，最后将子问题的解合并形成原问题的解。子问题本身也可以使用相同的方法：将它们分解成多个子问题，结果就形成了一个子问题的层次结构。到子问题简单到可以直接解决时，分解过程就可以停止了。

问题的层次结构反映了过程的层次。主过程作为执行的开始解决原始问题。为了解决原始问题，主过程调用许多解决子问题的过程，这些过程合并在一起就形成了原始问题的解。高层中的过程可以将它下一层中的过程当作抽象操作来使用。过程抽象提供每一层次支持的操作的视图，而这些操作如何由更低层次提供的操作去实现的方法却被隐藏了。

本节中的剩余部分着重讲述一个扩展的实例。我们使用过程分解原则来开发一个名为 `SortIntegerArgs` 的程序，它用一系列整型参数调用并且最后按照增序打印出这些参数。下面是程序的结果，黑体部分为用户输入值：

```
> java SortIntegerArgs 6 4 2 3 9 34 8 26 4 1 7
1 2 3 4 4 6 7 8 9 26 34
```

程序的过程分解可以用图2-2中的分层图来说明。每个过程都用一个命名的长方形框

表示，过程依赖的操作出现在与它相联系的下一层过程的长方形框中。通常，图中的每一层显示上一层中的过程所需要的操作，并且最高层显示整体要解决的问题。图2-2说明程序中的最高层过程（main）工作的三个步骤：

- 1) 将程序参数转变为整型数组。
- 2) 将数组a中的数进行排序。
- 3) 打印数组a。

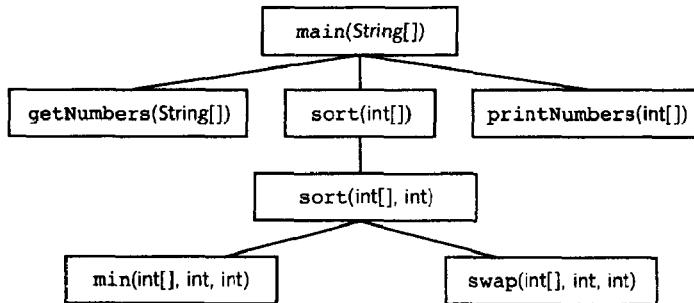


图2-2 程序SortIntegerArgs的结构

以下是main过程的定义：

```

// method of SortIntegerArgs class
public static void main(String[] args) {
    // EFFECTS: If args[i] is a badly formed integer for
    // some i, prints an error message; else prints the
    // ints denoted by args in nondecreasing order.
    try {
        int[] a = getNumbers(args);
        sort(a);
        printNumbers(a);
    } catch (NumberFormatException e) {
        String msg = "Error: argument " + e.getMessage();
        msg += " is badly formed.";
        System.out.println(msg);
        System.exit(0);
    }
}
  
```

main过程的作用子句解释main过程解决的原始问题。因为main是通过命令行调用的，所以程序的客户也就是用户。如果用户用一个或多个错误类型的参数调用程序时，程序会打印出错误消息并且退出。下面是表明这种行为的样例：

```

> java SortIntegerArgs 4 pear 32 tomato
Error: argument pear is badly formed.
  
```

过程main依赖于三个过程，下面我们就要看一下这三个过程。首先是过程getNumbers的定义：

```

// method of SortIntegerArgs class
static int[] getNumbers(String[] args)
    throws NumberFormatException {
    // REQUIRES: args is not null.
    // EFFECTS: If some args[i] is badly formed throws
    // NumberFormatException; else returns an int array
    // containing the integers denoted by args.
    int[] a = new int[args.length];
  
```

```

for (int i = 0; i < args.length; i++)
    a[i] = Integer.parseInt(args[i]);
return a;
}

```

如果`getNumbers`传入一个错误类型的参数，那么就会由`Integer.parseInt`方法产生一个`NumberFormatException`类型的异常，并且传播到过程`getNumbers`中。注意整型数组的值的顺序是和输入的`args`数组一样的，但这是一种人为的实现方法，`getNumbers`的说明没有保证这一点。

过程`printNumbers`也是由`main`过程调用的。定义如下：

```

// method of SortIntegerArgs class
static void printNumbers(int[] a) {
    // REQUIRES: a is not null.
    // MODIFIES: System.out
    // EFFECTS: prints the items in array a
    //           in the order a[0], a[1], ..., a[a.length-1]
    for (int i = 0; i < a.length; i++)
        System.out.print(a[i] + " ");
    System.out.println();
}

```

`main`过程执行的最有趣的任务是排序，就是将一系列可以比较的值按顺序排列的过程。在`SortIntegerArgs`程序中，是进行整型值排序。下面这个有一个参数的过程`sort`不同于一般的有两个参数的排序过程：

```

// method of SortIntegerArgs class
static void sort(int[] a) {
    // REQUIRES: a is not null.
    // MODIFIES: a
    // EFFECTS: Sorts array a in nondecreasing order.
    sort(a, a.length);
}

```

注意，`sort`过程需要由一个有效的数组引用来调用。这样做是合理的，因为我们开发的`sort`要在满足前置条件的情况下使用：当`SortIntegerArgs.main`调用`sort`时，可以知道它是以一个有效的数组引用来调用的。当然还可以定义一个更为一般的`sort`过程，它可以在任何环境下使用（如练习2.9和练习2.10）。类似的注释适用`getNumbers`和`printNumbers`这两个过程。

以下是有两个参数的`sort`过程说明：

```

// method of SortIntegerArgs class
static void sort(int[] a, int n)
    // REQUIRES: a is not null, and 0 <= n <= a.length.
    // MODIFIES: a
    // EFFECTS: sorts a[0..n-1] in nondecreasing order.

```

作用子句使用了符号 $a[0..n-1]$ 表示了数组 a 的前 n 个元素。更一般的表达式 $a[i..j]$ 表示长度为 $j - i + 1$ 的子数组，它包含了元素 $a[i], a[i+1], \dots, a[j]$ ，这里 $0 \leq i \leq j \leq a.length$ 。

排序可以使用多种方法。在有两个参数的`sort`过程的实现中，我们使用了选择排序(selection sort)。这种方法比其他的排序方法效率低一些，但是它很容易执行。选择排序法先扫描数组 $a[0..n-1]$ 并找出最小的一项。当找到时，将它与 $a[0]$ 交换，这时 $a[0]$ 即为最小的项， $a[1..n-1]$ 包含了剩下的项。下一次重复操作时，从 $a[1..n-1]$ 中找出最小项，并且将它与 $a[1]$ 交换。这时 $a[0..1]$ 包含了排序序列中最小的两项，并且 $a[2..n-1]$ 包含了剩余

的以任意顺序排列的项。我们继续以相同的方式重复刚才的操作，直到重复*n*次。当第*i*次重复操作完成后，*a[0..i-1]*包含了排序序列中最小的*i*个项，*a[i..n-1]*则包含了剩余的以任意顺序排列的项。*n*次重复操作之后，*a[0..n-1]*包含了*n*个排序序列中的项，任务就完成了。

为了执行选择排序，我们需要能够在子数组中找到最小项。我们可以使用2.3节中的过程min。下面我们采用一个过程min，从它的需求子句可以看出，它有着更强的前置条件：

```
// method of SortIntegerArgs class
static int min(int[] a, int lo, int hi) {
    // REQUIRES: Array a is not null,
    // and 0 <= lo <= hi < a.length.
    // EFFECTS: Returns the index of some smallest
    // element in a[lo..hi].
    int indx = lo;
    for (int i = lo+1; i <= hi; i++) {
        if (a[i] < a[indx])
            indx = i;
    }
    return indx;
}
```

选择排序过程还需要一个在数组内交换两项的过程：

```
// method of SortIntegerArgs class
static void swap(int[] a, int i, int j)
    // REQUIRES: 0 <= i,j < a.length.
    // MODIFIES: a
    // EFFECTS: Swaps the contents of a[i] and a[j]
```

描述了操作min和swap，我们就可以开始定义选择排序过程sort了：

```
// method of SortIntegerArgs class
static void sort(int[] a, int n) {
    // REQUIRES: Array a is not null and 0 <= n <= a.length.
    // MODIFIES: a
    // EFFECTS: Sorts a[0..n-1] in nondecreasing order.
    for (int i = 0; i < n; i++) {
        int indx = min(a, i, n-1);
        swap(a, i, indx);
    }
}
```

回顾本例，我们详细说明了一个问题，即将一系列整型参数按照一定的顺序排列并打印。我们将这个问题分解为三个任务：(1) 将程序参数转变为一个整型数组，(2) 将这些整型值进行排序，(3) 从左到右打印出整型数组的值。第一个和第三个任务可以分别直接由过程getNumbers和printNumbers完成。将一个整数数组排序被分解为更为一般的排序整数子数组，由两个参数的排序过程实现。这个任务又被分解为两个任务，一个是寻找子数组中的最小项，由过程min实现，另一个是将数组中的两项相交换，由过程swap完成。

练习

2.7 实现前面讨论的过程swap。

2.8 研究Java中的过程分解，将相关的过程声明为类方法是非常方便的（通过在每个过程头前加上关键字static），并且将它们放在同一个类中。例如，下面程序打印出直角三角形的边长，这些边长都是整型值：

```

public class RightTriangle {
    public static void main(String[] args) {
        if (args.length < 1) {

            String msg = "Usage: java RightTriangle n";
            System.out.println(msg);
            System.exit(0);
        }
        int n = Integer.parseInt(args[0]);
        printTable(n);
    }

    public static void printTable(int n) {
        for (int i = 1; i <= n; i++)
            for (int j = i+1; j <= n; j++) {
                double hyp = hypotenuse(i, j);
                if (hyp == Math.floor(hyp)) {
                    System.out.print(i + "\t" + j + "\t");
                    System.out.println((int)hyp);
                }
            }
    }

    public static double hypotenuse(int a, int b) {
        int c2 = square(a) + square(b);
        return Math.sqrt(c2);
    }

    public static int square(int k) {
        return k * k;
    }
}

```

RightTriangle程序是以最长的边长为参数来调用的，例如：

```

> java RightTriangle 20
3      4      5
5      12     13
6      8      10
8      15     17
9      12     15
12     16     20
15     20     25

```

使用本节中描述的类方法来实现SortIntegerArgs类，并且确保你的程序可以工作。

2.9 (重点) 根据下列说明写出过程sort:

```

static void sort(int[] a, int n) throws
    NullPointerException, IllegalArgumentException {
    // MODIFIES: a
    // EFFECTS: If a is null throws NullPointerException;
    //   else if 0 <= n <= a.length sorts a[0..n-1]
    //   in nondecreasing order;
    //   else throws IllegalArgumentException.
}

```

在SortIntegerArgs例子中使用的sort过程没有抛出异常——它的需求子句禁止以空引用或是非法参数调用。那么为什么在SortIntegerArgs程序环境下使用没有抛出异常的排序过程sort是可行的呢？

2.10 (重点) 使用以下说明编写过程sort:

```

static void sort(int[] a) throws
    NullPointerException {
    // MODIFIES: a
    // EFFECTS: If a is null throws NullPointerException;
    //   else sorts a in nondecreasing order.

```

2.11 下面过程min的实现实际上满足比文本中说明的更强的后置条件。这个过程可以用更强的后置说明, 如下所示:

```

static int min(int[] a, int lo, int hi)
    // REQUIRES: 0 <= lo <= hi < a.length.
    // EFFECTS: Returns the index of the smallest
    //   item in a[lo..hi] of least index.

```

但是正如文本给出的较弱后置条件, 只要求min过程返回最小项的下标值, 这是sort过程为了保证正确性面对main过程的全部要求。与此相似, 过程getNumber的实现满足一个比调用它的过程 (main过程) 所需要的要强的后置条件。从正确性的观点来看, 客户调用一个过程所实现的后置条件强于客户所要求的是否会产生问题? 从正确性的观点来看, 客户调用一个过程所需要的前置条件弱于客户所满足的会产生问题? (我们在5.5节中论述多态性和替代原则时会深入研究这些问题。)

2.12 实现一个名为SortStringArgs的程序, 它对字符串进行的操作类似于过程SortIntegerArgs对整型值做的操作。下面是程序执行和输出结果:

```

> java SortStringArgs twas brillig and the slithy toves
and brillig slithy the toves twas

```

2.5 递归

由于可以用其他低层次的抽象操作来实现抽象操作, 所以过程分解是可以实现的。有时候可能需要用相同的抽象操作来实现抽象操作自己。这种技术称为递归 (recursion), 一个调用自己一次或多次的过程是递归的。

我们可以用一个例子来说明这是可行的。可以给过程min实现的抽象操作建立一个递归执行, 即在子数组a[lo..hi]中查找最小项的位置。下面是我们的策略的伪代码:

```

if (lo==hi) then return lo;
else {
    best ← index of some smallest item in a[lo+1..hi];
    if (a[lo]<a[best]) then return lo;
    else return best;
}

```

在一般的情况下——当lo小于hi时——以上的伪代码这样描述这个过程: 在剩下的子数组 (即在a[lo+1..hi]中) 中寻找最小项, 然后返回这项或者a[lo]中较小者的下标值。在a[lo+1..hi]中寻找最小项的任务也可以由2.4节中对SortIntegerArgs程序定义的过程min来执行。这就导致以下新过程min2的实现:

```

static int min2(int[] a, int lo, int hi) {
    // REQUIRES: Array a is not null, and
    //   0 <= lo <= hi < a.length.
    // EFFECTS: Returns the index of some smallest
    //   element in a[lo..hi].
    if (lo == hi)
        return lo;
    else {

```

```

    int best = min(a, lo+1, hi);
    if (a[lo] < a[best]) return lo;
    else return best;
}
}

```

因为过程min2并没有调用它自身，所以它还不是递归的。下面的观察使我们可以将过程min2调整成为递归的：过程min和min2实现完全相同的抽象操作。因此else块中的第一个语句可以修改为：

```
int best = min2(a, lo+1, hi);
```

下面是过程min2的递归版本：

```

static int min2(int[] a, int lo, int hi) {
    // REQUIRES: Array a is not null, and
    // 0 <= lo <= hi < a.length.
    // EFFECTS: Returns the index of some smallest
    // element in a[lo..hi].
    if (lo == hi)
        return lo;
    else {
        int best = min2(a, lo+1, hi);
        if (a[lo] < a[best]) return lo;
        else return best;
    }
}

```

当一个问题可以被分解为相同类型的较小的子问题时就有可能使用递归。我们用楷体表示这个关键词。首先，子问题在这样的意义下必须小于原问题，即更接近于直接解决问题。当过程调用自身时，它必须朝着直接的解决方案前进。第二，子问题必须与原问题是相同的类型。因为这两个问题都是由相同的抽象操作解决的，解决原问题的过程也可以用于解决较小的子问题。

为了确定问题是否小到可以直接解决，过程会测试终止条件（stopping condition）是否得到满足。一旦终止条件得到了满足，这个问题就可以直接解决而无需进一步的递归调用。将这些想法用于递归过程min2，我们可以发现：

- 终止条件是 $lo == hi$ 。这个问题是在只剩一个元素时可以直接解决。
- $lo < hi$ 是递归条件。当 $lo < hi$ 时，这个问题的元素个数为 $hi - lo + 1$ ，即将要处理的子数组的长度。过程向满足终止条件的问题前进。

当抽象操作可以用它自身来定义时，就可以使用递归。这样的定义称为递归定义（recursive definition）或是归纳定义（inductive definition）。考虑著名的用非负整数定义的阶乘函数：

$$n! = n(n-1)(n-2) \cdots 3 \times 2 \times 1 \times 1$$

阶乘函数也可以用递归定义来计算，如下所示：

$$n! = \begin{cases} 1 & n = 0 \\ n(n-1)! & n > 0 \end{cases}$$

注意到在这种定义下，我们有 $0! = 1$ ，这个递归定义产生了下面 $4!$ 的推导：

$$\begin{aligned}
 4! &= 4 \times 3! \\
 &= 4 \times 3 \times 2! \\
 &= 4 \times 3 \times 2 \times 1!
 \end{aligned}$$

```
=4 × 3 × 2 × 1 × 0!
=4 × 3 × 2 × 1 × 1
=24
```

这个阶乘的递归调用说明了 $n!$ 的递归实现，它的终止条件是 $n==0$ ，它的递归条件是 $n>0$ 。递归过程如下：

```
static int factorial(int n) {
    // REQUIRES: n is nonnegative.
    // EFFECTS: Returns the factorial of n.
    if (n == 0)
        return 1;
    else
        return n * factorial(n-1);
}
```

过程 `factorial` 要求它的参数为非负整数。当然，也可以修改说明来去掉这个前置条件，却要增加一个后置条件，即当过程 `factorial` 用一个负数调用时就抛出一个异常 `IllegalArgumentException`。做到这一点的最有效的方法是定义一个对说明作了修改的新过程：

```
static int fact(int n) throws IllegalArgumentException {
    // EFFECTS: If n<0 then throws IllegalArgumentException;
    //   else returns the factorial of n.
    if (n < 0) throw new IllegalArgumentException();
    return factorial(n);
}
```

过程 `fact` 不是递归的，但是它调用的过程 `factorial` 是递归的。用一个负数来调用过程 `factorial` 是不可能的，如果输入 n 是负值时，`fact` 会抛出一个异常。因此当 `fact` 调用 `factorial` 时，它能保证 `factorial` 的需求子句肯定会得到满足。像 `fact` 这样的非递归过程，为一个递归过程提供正确的输入值，被称为非递归处理程序（nonrecursive shell）。

有时候你必须实现一个没有递归定义的抽象操作，这时你可以自己构造一个。如下面的抽象操作，计算整数 x 在整型子数组 $a[lo..hi]$ 中出现的次数：

```
static int count(int x, int[] a, int lo, int hi)
// REQUIRES: Array a is not null, and
//   either 0 <= lo <= hi < a.length, or lo > hi.
// EFFECTS: If lo <= hi returns the number of times
//   that x occurs in a[lo..hi]; else returns 0.
```

当跟踪 x 出现的次数时，基于迭代的实现需要遍历子数组 $a[lo..hi]$ ，像下面的实现：

```
static int count(int x, int[] a, int lo, int hi) {
    int count = 0;
    for (int i = lo; i <= hi; i++)
        if (a[i] == x)
            count++;
    return count;
}
```

相反，一个递归的实现可以按如下的递归定义进行。让 $C_{i,j}$ 表示 x 在子数组 $a[i..j]$ 中出现的次数，可以得到：

$$C_{i,j} = \begin{cases} 0 & i > j \\ C_{i+1,j} & i \leq j \text{ 且 } x \neq a[i] \\ 1 + C_{i+1,j} & i \leq j \text{ 且 } x = a[i] \end{cases}$$

这个定义从上到下包含了三种情况：第一种情况是子数组 $a[i..j]$ 为空时， x 根本不存在；第二种情况下，子数组不为空，但是因为 x 不等于 $a[i]$ ，所以 x 在 $a[i..j]$ 中出现的次数等于 x 在 $a[i+1..j]$ 中出现的次数；第三种情况是 x 等于 $a[i]$ ，所以 x 在 $a[i..j]$ 中出现的次数等于 1 加上 x 在 $a[i+1..j]$ 中出现的次数。这个递归定义产生了下面的过程 `count` 的递归实现：

```
static int count (int x, int[] a, int lo, int hi) {
    if (lo > hi)
        return 0;
    else if (x != a[lo])
        return count(x, a, lo+1, hi);
    else
        return 1 + count(x, a, lo+1, hi);
}
```

递归是一种强大的方法，在后面的章节中会用到它。而且我们也会用到递归结构（recursive structure），它是一种用自身定义的数据结构。

练习

2.13 在 2.4 节中的定义的两个参数的 `sort` 过程中，将调用的过程 `min` 替换为过程 `min2`，然后重新编译运行过程 `SortIntegerArgs`，程序仍然是正确的吗？应该是正确的，因为过程 `min` 和 `min2` 实现了相同的抽象操作。

2.14 幂函数（power function）用下面的递归方式定义，其中 b 是非负整数：

$$a^b = \begin{cases} 1 & b = 0 \\ a \cdot a^{b-1} & b > 0 \end{cases}$$

例如，下面是 2^3 的推导：

$$\begin{aligned} 2^3 &= 2 \times 2^2 \\ &= 2 \times 2 \times 2^1 \\ &= 2 \times 2 \times 2 \times 2^0 \\ &= 2 \times 2 \times 2 \times 1 \\ &= 8 \end{aligned}$$

基于 a^b 的这种递归定义，编写一个过程 `power` 的递归实现：

```
static int power(int a, int b)
    // REQUIRES: b is nonnegative.
    // EFFECTS: Returns a raised to the power b.
```

2.15 如果执行一个递归过程，并且没有任何终止条件得到满足，那么这一系列的递归调用将会永远执行并且过程永不会结束。事实上，如果合法输入导致了递归调用的无止境调用，那么这个递归过程是错误的。用 Java 编写一个无论任何输入都永远不会终止的递归过程，再编写一个终止于某些输入但不会终止于所有输入的递归过程。

2.16 斐波纳契数列（Fibonacci sequence）是这样开始的：

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, …

前两个元素是 0 和 1，并且后面的每一个元素都等于前面的两个元素之和。下面是它的递归定义：

$$fib(n) = \begin{cases} n & n = 0 \text{ 或 } n = 1 \\ fib(n-1) + fib(n-2) & n > 1 \end{cases}$$

定义一个基于刚才给出的递归定义的递归过程来计算斐波纳契数列中的元素：

```
static long fib(int n)
// REQUIRES: n is nonnegative.
// EFFECTS: Returns element n in the Fib sequence.
```

在一个Java程序中使用你的过程，用一个非负整数参数n，并且打印出它的斐波纳契序列数：

```
> java Fib 6
Fib(6) = 8
```

如果严格按照前面给出的递归定义，你会发现过程fib的递归版本效率很低。问题在于很多代价高的计算被重复执行。例如，过程调用fib(40)导致了调用fib(39)和fib(38)；调用fib(39)导致了调用fib(38)和fib(37)。可以看到调用fib(38)发生了两次。相同种类的重复在计算过程中出现过很多次。过程fib的递归版本的运行时间是输入值n的指数，所以时间消耗是相当大的，即便n非常小也是如此。

从这个经验中，我们不能得出递归本身是低效的这样的结论，而是我们这里使用的方法是低效的。下面是一个高效的递归方法（运行时间与输入值n成比例）：

```
static long fib(int n) {
// REQUIRES: n >= 0.
// EFFECTS: Returns element n in
// the Fibonacci sequence.
return fibHelp(n, 0, 1);
}

static long fibHelp(int n, long a, long b) {
// REQUIRES: a and b are successive elements in the
// Fibonacci sequence.
// EFFECTS: Returns nth successor to a in the
// Fibonacci sequence.
if (n == 0)
return a;
else
return fibHelp(n-1, b, a+b);
}
```

在这个实现中，非递归处理程序fib调用了递归过程fibHelp。编写过程fib版本，并且用它计算元素100的斐波纳契数列，并且使用你的fib递归版本来计算fib(100)。哪一个版本更快呢？

2.17 编写一个递归过程来执行下面的抽象操作：

(a) 计算整型子数组a[lo..hi]的和：

```
static int sumOver(int[] a, int lo, int hi)
// REQUIRES: Array a is not null, and
// 0 <= lo <= hi < a.length.
// EFFECTS: Returns the sum a[lo]+..+a[hi].
```

(b) 用整型数组a[lo..hi]中的值的平方来替代数组中的值：

```
static void squareMap(int[] a, int lo, int hi)
// REQUIRES: Array a is not null, and
// 0 <= lo <= hi < a.length.
// MODIFIES: a
// EFFECTS: Replaces each item x in a[lo..hi] by x2.
```

(c) 计算从1到n的所有正整型值的和：

```

static int summation(int n)
// REQUIRES: n is nonnegative.
// EFFECTS: Returns 0+1+...+(n-1)+n
(看看你能否想出一个在固定时间内设计出计算 $1+2+\dots+n$ 的和的公式。)
(d) 指出 $x$ 在 $a[lo..hi]$ 中首次出现的位置值, 或者是-1(子数组不出现 $x$ 的情况):
static int find(int x, int[] a, int lo, int hi)
// REQUIRES: Array a is not null, and
// 0 <= lo <= hi < a.length.
// EFFECTS: Returns the position of first x in
// a[lo..hi];
// returns -1 if x does not occur in a[lo..hi].

```

2.18 归并排序 (Merge sort) 是另一个用于排列整型数组中的值的算法。抽象操作如下:

```

static void msort(int[] a, int lo, int hi)
// REQUIRES: a is not null, and
// 0 <= lo <= hi < a.length.
// MODIFIES: a
// EFFECTS: Sorts a[lo..hi] in nondecreasing order.

```

用过程`msort`可以很容易实现数组排序。例如, 2.4节中带有一个参数的过程`sort`可以用归并排序实现如下:

```

static void sort(int[] a) {
// REQUIRES: a is not null.
// MODIFIES: a
// EFFECTS: Sorts array a in nondecreasing order.
msort(a, 0, a.length-1);
}

```

为了完成我们的工作, 必须实现`msort`过程, 以下是这个过程如何工作的伪代码描述:

```

if (lo==hi) then return;
else {
    mid ← (lo + hi) / 2;
    sort a[lo..mid] in nondecreasing order;
    sort a[mid+1..hi] in nondecreasing order;
    merge(a, lo, mid, hi);
}

```

在一般情况下, 当 $lo < hi$ 时, 这种想法是将子数组 $a[lo..hi]$ 分成长度大致相同的两部分, 然后单独(并且使用递归)将两部分排序, 最后将已经排序的两部分合并:

```

static void merge(int[] a, int lo, int mid, int hi)
// REQUIRES: a is not null;
// 0 <= lo <= mid < hi < a.length;
// a[lo..mid] and a[mid+1..hi] are both sorted in
// nondecreasing order.
// MODIFIES: a
// EFFECTS: Sorts a[lo..hi] in nondecreasing order.

```

过程`merge`将两个排过序的子数组 $a[lo..mid]$ 和 $a[mid+1..hi]$ 合并为一个排序的数组 $a[lo..hi]$: 它通过用两个变量跟踪两个数组的下标, 从数组最左边的位置开始, 依次比较两个元素的大小。在每一次比较中, 正在被检索的两项相比较, 将较小的一项复制到临时数组`b`中, 并且它的下标加1。一旦其中一个下标超过了数组中最右边元素的下标, 数组中剩余的元素就会被复制到数组`b`中。下面是过程`merge`的伪代码实现:

```
static void merge(int[] a, int lo, int mid, int hi) {
```

```

int[] b = new int[hi-lo+1];
int i = lo,      // index into a[lo..mid]
    j = mid+1, // index into a[mid+1..hi]
    k = 0;      // index into b
while (i < mid+1 and j < hi+1) {
    if (a[i] < a[j])
        b[k++] = a[i++]; // copy a[i] into b
    else
        b[k++] = a[j++]; // copy a[j] into b
}
if (i==mid+1) then copy a[j..hi] into b[k..hi-lo];
else copy a[i..mid] into b[k..hi-lo];
copy b into a[lo..hi];
}

```

实现过程`msort`和`merge`。然后编写一个Java程序，练习一下新的排序方法（你可以修改程序`SortIntegerArgs`的实现，这样就可以使用合并排序）。为了执行过程`merge`，你可以使用`java.lang.System.arraycopy`方法来复制子数组。

小结

可以用两种不同的方式来理解一个过程：一是作为一种抽象操作，将输入映射到输出和副作用，二是作为实现操作过程的说明。过程抽象将过程看作是一种抽象操作。这种观点最适合调用过程的客户和这些客户程序的编写者。过程抽象的主要优点是客户可以将一个过程看成一个操作而忽略它的实现细节。

过程的前置条件是调用过程的客户必须满足的条件。过程的后置条件是在客户满足了过程的前置条件的情况下，过程必然会出现的操作的条件。满足过程的前置条件是客户的应尽义务；而过程的后置条件是由过程来保证的。

为了说明由过程实现的操作，我们用需求子句来描述过程的前置条件，用作用子句来描述过程的后置条件，它们都出现在紧随过程头后的说明注释中。除此之外，修改子句列出了那些状态可能被过程改变的对象。过程可以使用断言来确定它的需求子句和作用子句是否得到了满足。Java也提供了能够抛出异常、捕捉异常和处理异常的异常机制。

当我们运用过程分解的方法来构造程序时，我们可以欣赏到过程抽象的强大功能。问题的解决方法采用了过程的层次结构的形式，每一层的过程都提供了上一层需要的操作。最高层的过程代表了整体解决原问题的操作。当我们定义自己调用自己的递归过程时，也需要依赖过程抽象。一个递归过程恰恰代表了它自己需要的操作。通过将一个递归过程作为抽象操作，我们可以理解一个调用自己的过程。

和每一个对象相关联的是一组过程，即对象的方法。这些方法代表对象已定义的一组行为。通过把这些方法视为抽象操作，我们可以利用这些行为来观察这个对象而且忽略它的实现。因此，过程抽象可以作为数据抽象的主要基石。数据抽象将是下一章的主题。

第3章 数据抽象

在上一章中，我们学习了过程抽象，即为何一个过程被抽象地看作一个操作。客户把调用一个过程看作一个操作，而并不需要知道这个操作是如何实现的。在这一章中，我们将研究另一种不同类型的抽象类型，它是对象模型的核心，这就是数据抽象（data abstraction）。在数据抽象模式下，一个数据值，比如一个对象，不仅是一堆数据，还是紧密相关的一组操作的集合和灵活方便地使用这些操作的协议。客户通过和抽象数据的操作交互来实现访问数据，而并不需知道它所访问的数据是如何构造的或操作是如何实现的。抽象数据提供给客户一个公共接口——一组操作——但隐藏它的内部实现。

3.1节介绍数据抽象的基本概念和它的主要优点。3.2节将通过表示平面上点和矩形的两个新类，来介绍数据抽象具体设置。3.3节讨论封装和信息隐藏。3.4节概括介绍如何运用Java 2D制作计算机图形。3.5节将把这些技术运用到一个程序中，这个应用程序可以在屏幕的窗口中画矩形。尽管这个应用程序对于它所产生的图形来说很简单，但它却为本书后面出现的其他计算机图形应用程序提供一个模板。

3.1 抽象数据类型

一个具体数据类型（concrete data type）是一个数据值（data value）的集合，包括具体的表示方法和一组操作。Java提供很多种原始的具体数据类型：数值型，字符型和布尔型。例如，Java的整型数据类型是 -2^{31} （2147483648）到 $2^{31}-1$ （2147483647）范围上所有整数的集合，一个整数用4个字节的有符号二进制补码存储表示，并提供加、减、乘、赋值等很多种操作。

显然，使用整数时，你并不需要知道它用4个字节有符号二进制补码表示，或是其他类似的整数表示方法；你也不知道整数的操作是如何实现的；仅需的知识是如何使用它们提供的操作。例如，可以用整数赋值和加操作将两个整数相加，并把结果赋给一个整型变量：

```
int i;  
i = 6 + 7;
```

整数是如何存储的及具体的操作过程如何实现与整数的使用无关（虽然如此，Java还是提供了几个位运算符以便于整数进行位操作运算）。

字符串是另一种我们比较熟悉的数据类型。像整型一样，你可能已经知道如何使用字符串，而不知道它在内部是如何表示的或者串操作是如何实现的。下面看一个简单使用字符串的Java程序，这个程序以相反顺序连接它的输入字符串参数，并且将结果转换为大写字母输出。

```
public class SillyEcho {  
    public static void main(String[] args) {  
        String s = "";  
        for (int i = 0; i < args.length; i++)
```

```

    s = args[i] + " " + s;
    System.out.println(s.toUpperCase());
}
}

```

以下是运用该程序的例子，粗体字是键入的命令行，而普通字体是程序运行结果。

```

> java SillyEcho Hello Goobly World
WORLD GOOBLY HELLO

```

为了编写SillyEcho程序，你必须知道如何运用具体的字符串操作：运用 + 操作符来连接两个字符串；在一串字符两边加上双引号（如"apple"）表示一个字符串；利用String类的toUpperCase方法把一个字符串转换成大写字符串。这些操作都可直接使用，不需知道字符串是如何表示的。事实上在Java中，字符串可以依据不同的需要表示成不同形式，这里不考虑这些问题。

通常来说，具体数据类型可以根据数据值的结构和行为进行分类，结构是指它的内部表示，行为是指它的使用方法。使用数据时，对于它的结构和行为而言，我们会更关心它的行为。例如，对两个整数进行加或乘运算时，并不关心它们在计算机中如何存储；同样，连接两个字符串时并不需知道它们的内部表示。抽象数据类型概念的提出正是基于这种对数据值上的操作的关注。抽象数据类型（abstract data type），简称数据类型（data type），是一组数据和在其上的一组操作。所谓抽象是指只需知道数据如何使用，而不需要知道数据的内部表示和实现方法。数据抽象（data abstraction）——通过运用抽象数据类型实现——介绍如何表达数据的操作，而忽略其具体的表示和实现。

在数据抽象中，一个数据值被表示成一组数据和一组公共操作，这些操作构成这些数据的接口（interface），客户正是通过接口操作数据的。而数据值的实现（implement）包括它的内部表示和基于这些表示的操作的实现。这样数据抽象仅提供给客户数据值的接口而屏蔽了它的实现。

使用数据抽象有很多优点。首先，客户不需要了解详细的实现细节就可使用它。我们可以直接使用整数和字符串，而无须知道这些数据类型的实现。其次，由于对客户屏蔽了数据类型的实现，因此只要保持接口不变，数据实现的改变并不影响客户的使用。最后，由于数据类型接口规定了客户与数据之间所有可能的交互，因此可以把数据类型理解为展示已定义好的行为的模块。这样我们的思想就上升到由接口提供的抽象概念层。在这一章中我们将会详细讨论数据抽象的这些优点和更多其他的优点。

除了数字、字符和布尔型等原始数据类型外，Java还提供更广泛的预定义的复合数据类型（compound data type）：Java语言标准类。由于Java中的对象通过引用被处理，所以复合数据类型有时也被称为引用类型（reference type）。当一个数据值是某个类的实例时，我们通常把它作为一个实例（instance）或对象（object）来引用。我们把"hello"作为一个String类的实例或String对象的引用，也可简单地称为字符串的引用。

除了提供字符串类型和整型，Java还提供构造新的复合数据类型的方法。这是通过运用Java的数组、接口以及类来实现的。接口用来定义新的数据类型但是没有实现它们，而类则既定义又实现了新的数据类型。下面我们将先讨论类；而把接口放到本书的后面部分讨论。

3.2 说明和实现数据抽象

本章将通过开发两个新类——分别表示平面上点和矩形——来探讨数据抽象。我们将

使用前一章讨论过的过程说明注释方法描述一个类的行为。特别是每个方法都用注释描述它的需求、修改和额外作用。

3.2.1 点

笛卡儿平面有两个在原点相交的互相垂直的轴：水平伸展的x轴和垂直伸展的y轴。每一个点的位置由一个形如 (x, y) 的整数对给定，并表明如何从原点到达该点：从原点开始，先沿x轴移动 x 个单位，然后沿y轴平行的移动 y 个单位。如果 x 是正整数，则沿x轴的正半轴（向右）移动；如果是负整数，则沿x轴负半轴（向左）移动。 y 的情况与此类似。点 $(0, 0)$ 被称作原点（origin），参见图3-1（你可能会对x轴正半轴向右，y轴正半轴向上的坐标系比较熟悉。但是这里描绘的类似图3-1平面坐标是Java的默认绘图坐标系。在本章的后面会详细说明）。

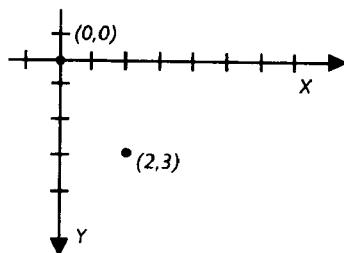


图3-1 笛卡儿平面和点 $(2, 3)$

我们将开发一个用来处理平面中的点的类PointGeometry。从哪里开始呢？切记在考虑如何实现一个数据类型以前，必须先理解它，也就是说，要先把它作为一个抽象数据类型来理解。这个类型支持什么操作？这些操作如何在一起使用？先来看一下类PointGeometry的类框架结构（class skeleton），它可以帮助我们理解相应的数据类型：

```
public class PointGeometry {

    public PointGeometry(int x, int y)
        // EFFECTS: Initializes this to the point (x,y).

    public PointGeometry(PointGeometry p)
        throws NullPointerException
        // EFFECTS: If p is null throws NullPointerException;
        // else initializes this to (p.getX(),p.getY()).

    public PointGeometry()
        // EFFECTS: Initializes this to the origin (0,0).

    // x coordinate property
    public int getX()
        // EFFECTS: Returns x coordinate.

    public void setX(int newX)
        // MODIFIES: this
        // EFFECTS: Changes x coordinate to newX.

    // y coordinate property
    public int getY()
```

```

// EFFECTS: Returns y coordinate.

public void setY(int newY)
    // MODIFIES: this
    // EFFECTS: Changes y coordinate to newY.

// other methods
public double distance(PointGeometry p)
    throws NullPointerException
    // EFFECTS: If p is null throws NullPointerException;
    // else returns the distance between p
    // and this point.

public java.awt.Shape shape()
    // EFFECTS: Returns this point's shape.

public void translate(int dx, int dy)
    // MODIFIES: this
    // EFFECTS: Translates this point by dx along x
    // and dy along y:
    // this_post.getX() == this.getX()+dx and
    // this_post.getY() == this.getY()+dy.

public boolean equals(Object obj)
    // EFFECTS: Returns true if obj is a PointGeometry
    // and obj.getX()==getX() and obj.getY()==getY();
    // else returns false.

public String toString()
    // EFFECTS: Returns the string "(x,y)".
}

```

尽管一个类框架结构看起来像Java源代码，但是它并不是正式语言结构，比如类框架结构不可以编译。这里用类似Java的语法表达类框架结构是为了更好地解释类中的方法，并没有真正实现它。当然你可以把类框架结构当作描述新的数据类型的自定义结构。

一个类框架结构描述如何使用一个类。通常通过一个使用类的程序可以更容易、更好地理解类。下面将要看到的TryPoint类就是一个使用PointGeometry类的应用程序。该程序有四个参数：第一对整型参数是点 p_1 的 x 和 y 坐标，第二对整型参数是点 p_2 的 x 和 y 坐标。程序打印出点 p_1 、点 p_2 和它们之间的距离，以及点 p_1 按 p_2 的坐标移动后 p_1 的新位置。如果 p_2 没有输入，则默认 p_2 为原点 $(0, 0)$ 。下面是两个执行例子：

```

> java TryPoint 3 4 -5 8
point 1: (3,4)
point 2: (-5,8)
distance: 8.94427190999916
point 1 translated: (-2,12)

> java TryPoint 3 4
point 1: (3,4)
point 2: (0,0)
distance: 5.0
point 1 translated: (3,4)

```

以下是TryPoint类的定义：

```

public class TryPoint {
    public static void main(String[] args) {
        if ((args.length != 2) && (args.length != 4)) {
            System.out.println("USAGE: java TryPoint x1 y1 [x2 y2]");
        }
    }
}

```

```

        System.out.println(msg);
        System.exit(0);
    }
    int x1 = Integer.parseInt(args[0]);
    int y1 = Integer.parseInt(args[1]);
    PointGeometry p1 = new PointGeometry(x1, y1);
    PointGeometry p2 = new PointGeometry();
    if (args.length == 4) {
        int x2 = Integer.parseInt(args[2]);
        int y2 = Integer.parseInt(args[3]);
        p2.setX(x2);
        p2.setY(y2);
    }
    System.out.println("point 1: " + p1);
    System.out.println("point 2: " + p2);
    System.out.println("distance: " + p1.distance(p2));
    p1.translate(p2.getX(), p2.getY());
    System.out.println("point 1 translated: " + p1);
}
}

```

当你用如下命令行调用TryPoint程序时：

```
> java TryPoint 3 4 -5 8
```

程序从静态方法TryPoint.main开始执行。记住当你运行一个Java程序时，必须提供一个类的名字（本例中的TryPoint），然后系统找到类中的名为main的方法并从它开始执行。main方法声明一个String[]类型的参数，系统通过这个字符串数组把参数传递给main（本例中参数名是args）。由于该参数定义为字符串，本例中需要的是整型数，因此在使用它们之前须转化为整数。下面语句：

```
int x1 = Integer.parseInt(args[0]);
```

把第一个参数转为整数，并将结果赋给变量x1。注意点p2被构造时，它是在原点（0, 0）上，但如果程序有第二对参数时，则点p2用下面命令改变位置到（x2, y2）：

```
p2.setX(x2);
p2.setY(y2);
```

现在已经了解如何使用点数据类型，下面将注意力转到如何实现它的问题上。需要先确定两件事情：第一，为PointGeometry确定一种表示法，这里选择用点的x和y坐标值来代表点，并将这些值保存在具有相同名字的实例域中：

```
// fields of PointGeometry class
protected int x, y;
```

第二，类中的方法的实现要依照选择的存储结构而定。先来看PointGeometry的构造器，第一个构造器有两个整型参数并将它们赋给类域x, y：

```
public PointGeometry(int x, int y) {
    this.x = x;
    this.y = y;
}
```

在上面的构造器中，关键字this指正在构造的对象，通过它访问对象域，this.x指这个对象的实例域x，而赋值语句后的x指方法的参数x。

第二个构造器的参数是一个点对象（PointGeometry）；

```
public PointGeometry(PointGeometry p)
```

```

    throws NullPointerException {
    this(p.getX(), p.getY());
}
}

```

在前面构造器中，关键字this用来调用第一个构造器。由this调用构造器时，具体调用哪一个构造器由参数的数量和类型决定。在本例中，this以两个整数参数调用，由于第一个构造器的参数和它相符，因此第一个构造器被调用。顺便提一下，使用this调用其他构造器时，这一语句必须是构造器的第一条语句。

最后一个构造器没有参数，并且构造一个表示原点的新点：

```

public PointGeometry() {
    this(0, 0);
}
}

```

下面来看访问和改变点x坐标的方法。用来返回特性值的方法称作获取者（getter）。获取者getX返回点的x坐标特性，保存在点的x域中：

```

// method of PointGeometry class
public int getX() { return x; }

```

用来改变特性值的方法称作设置者（setter）。设置者setX用来改变点的x坐标值：

```

// method of PointGeometry class
public void setX(int newX) { x = newX; }

```

类中既有获取者又有设置者的性质称作类的特性（property）。根据惯例，获取者和设置者的名字是在大写特性名前加上get和set，例如类特性x用getX和setX命名获取者和设置者。一个特性的获取者和设置者方法在一起称为存取器（accessor）。

从广义上讲，设置者方法是一种称为增变方法（mutator method），简称为增变器（mutator）。所有改变对象实例域的方法都称为增变器，也就是说可以改变对象状态的任何方法都称为增变器。同样，获取者方法广义上属于选择器（selector），选择器指所有访问对象实例域却不改变它们的值的方法。由此得出结论，设置者是增变器，被用来改变某个特性的值，获取者是选择器，被用来访问某个特性的值。

注意一个类的特性不需要和它的域一致是很重要的。在PointGeometry类中，一个点的x坐标实际上保存在具有相同名字的域上。然而，在下一节中，我们将看到一个类的例子，它的特性没有被保存在域中，而是通过计算被导出。特性的值并不是必须要被保存在类的一个域中，并且特性通常不需要特殊的方法来实现。从客户的角度看，类的特性是类的抽象的一部分，并且从属于类的接口，而不是它的实现。

PointGeometry类也定义了访问特性y，y特性的方法定义与x特性的类似：

```

// methods of PointGeometry class
public int getY() { return y; }
public void setY(int newY) { y = newY; }

```

定义了PointGeometry的构造器和特性x和y的存取器后，接下来看一下其余的方法。distance方法获得输入点p，返回p到该点间的距离。两点之间的距离可根据勾股定理计算：两点之间的线段被看作是一个直角三角形的斜边，该直角三角形的两条直角边分别和x轴和y轴平行。斜边的长等于直角边的平方和的平方根。以下是distance的定义：

```

// method of PointGeometry class
public double distance(PointGeometry p)
    throws NullPointerException {
    long dx = this.getX() - p.getX();
    long dy = this.getY() - p.getY();
}

```

```

        double d2 = dx * dx + dy * dy;
        return Math.sqrt(d2);
    }

```

shape方法返回一个**java.awt.Shape**类型的对象。在3.4节中将看到它的用法，下面是**shape**方法的实现：

```

// method of PointGeometry class
public java.awt.Shape shape() {
    return new Ellipse2D.Float(getX()-2, getY()-2, 4, 4);
}

```

translate方法使点沿x轴水平移动dx个单位，沿y轴水平移动dy个单位。如果dx为正，则点向右平移，否则向左平移；dy的移动与此类似。下面是实现：

```

// method of PointGeometry class
public void translate(int dx, int dy) {
    setX(getX() + dx);
    setY(getY() + dy);
}

```

如果两个点的x和y特性值都相同，则它们被认为相等。**equals**方法用来比较两个点是否相等：

```

// method of PointGeometry class
public boolean equals(Object obj) {
    if (obj instanceof PointGeometry) {
        PointGeometry p = (PointGeometry) obj;
        return (p.getX() == getX() && (p.getY() == getY()));
    }
    return false;
}

```

由于**equals**方法的参数可以以任何类型的对象被调用，所以该方法先要用Java内置的**instanceof**操作符测试该参数是否是**PointGeometry**对象，如果是，就把该参数强制转换成**PointGeometry**类型：

```
PointGeometry p = (PointGeometry) obj;
```

由于先用**instanceof**测试**obj**是**PointGeometry**对象引用，因而强制转换是合法的。然后把点p的x和y特性值与当前点的特性值比较，如果相等则返回**true**，否则返回**false**。如果参数**obj**不是**PointGeometry**对象，**equals**方法返回**false**。

toString方法返回一个表示点的字符串，使用我们熟悉的有序对表示法。例如，点(5, 7)用字符串"(5, 7)"表示：

```

// method of PointGeometry class
public String toString() {
    return "(" + getX() + "," + getY() + ")";
}

```

当某个地方需要把点对象转化成字符串时，**toString**方法会被自动调用。例如，假设变量p1引用一个点对象，表达式

```
"point 1: " + p1
```

具有混合类型：操作符+的左操作数是字符串，右操作数是点对象。左操作数的类型(**String**)通知编译程序，+操作符在上下文中包括一个字符串连接，由于右操作数是一个点对象，而不是一个字符串，所以它的**toString**方法被自动调用来转换成一个字符

串，则整个表达式的值是字符串，表示为“point 1: (5,7)”。

练习：

3.1（重点）考虑上面给出的distance、equals、shape、translate和toString方法的实现，每一个实现都避免直接引用PointGeometry类的x和y域，而是间接利用getX和getY方法来获得点的坐标。例如，toString方法可以用下面语句实现：

```
public String toString() {
    return "(" + x + "," + y + ")";
}
```

请思考一下，以避免直接引用类的域的方法实现方法的优点和缺点各是什么？

3.2（重点）Range对象用来表示一段实数的范围，它由两个整数特性决定：最小值（min）和最大值（max）特性，最小值不能大于最大值。x的范围 $2 \leq x \leq 6$ 可以被写成[2..6]，这里2是最小值，6是最大值。范围的长度（length）是最大值和最小值之差（6-2=4），范围的大小（size）是它所包含的整数的个数（5）。以下是一个Range类的框架结构：

```
public class Range {

    public Range(int min, int max)
        throws IllegalArgumentException
    // EFFECTS: If max < min throws
    //    IllegalArgumentException; else initializes
    //    this range to [min..max].

    public Range(Range r) throws NullPointerException
    // EFFECTS: If r is null throws
    //    NullPointerException; else initializes this
    //    to the range [r.getMin()..r.getMax()].
}

public Range()
    // EFFECTS: Initializes this range to [0..0].

public int getMin()
    // EFFECTS: Returns this range's min.

public void setMin(int newMin)
    throws IllegalArgumentException
    // MODIFIES: this
    // EFFECTS: If getMax() < newMin throws
    //    IllegalArgumentException; else sets min
    //    to newMin: this_post.getMin() == newMin.

public int getMax()
    // EFFECTS: Returns this range's max.

public void setMax(int newMax)
    throws IllegalArgumentException
    // EFFECTS: If newMax < getMin() throws
    //    IllegalArgumentException; else sets max
    //    to newMax: this_post.getMax() == newMax.

public void setMinMax(int newMin, int newMax)
    throws IllegalArgumentException
    // MODIFIES: this
    // EFFECTS: If newMax < newMin throws
    //    IllegalArgumentException; else updates
```

```

// min and max: this_post.getMin() == newMin
// and this_post.getMax() == newMax.

public int length()
    // EFFECTS: Returns this range's length: max-min

public int size()
    // EFFECTS: Returns this range's size: length() + 1.

public boolean contains(int a)
    // EFFECTS: Returns true if getMin() <= a <= getMax();
    // else returns false.

public boolean equals(Object obj)
    // EFFECTS: Returns true if obj is a Range and
    // obj.getMin() == getMin() and
    // obj.getMax() == getMax(); else returns false.

public String toString()
    // EFFECTS: Returns the string "[min..max]".
}

```

定义一个Range类，它实现以上给定的说明：类应该定义一个存储结构来表示范围，并且根据选择的存储结构来实现类框架结构所声明的方法。你可以用下面的程序测试类。这个程序要求你考虑0到100之间的数，然后提出一系列的问题来逐渐减小数字范围，它使用Range对象来跟踪坐标的当前范围。程序开始时，初始范围是[0..100]（尽管如果以整数参数调用该程序其最大值可能超过100）。每回答一次问题范围的大小减少一半，直到范围只包括一个数为止。这种重复不断一分为二减小集合大小的过程称为二分搜索法（binary search）。

```

public class GuessNumber {
    public static void main(String[] args) {
        int n = 100;
        if (args.length > 1) {
            System.out.println("USAGE: java GuessNumber [n]");
            System.out.println(msg);
            System.exit(1);
        } else if (args.length == 1)
            n = Integer.parseInt(args[0]);
        Range rng = new Range(0, n);
        ScanInput in = new ScanInput();
        System.out.print("think of a number ");
        System.out.println("between 0 and " + n);
        String response;
        try {
            while (rng.size() > 1) {
                System.out.println("current range: " + rng);
                int mid = (rng.getMin() + rng.getMax()) / 2;
                System.out.print("is your number greater ");
                System.out.println("than " + mid + "?");
                response = in.readString();
                if (response.charAt(0) == 'y')
                    rng.setMin(mid + 1);
                else
                    rng.setMax(mid);
            }
            System.out.print("you're thinking of ");
            System.out.println(rng.getMin() + "!");
        }
    }
}

```

```
    } catch (IOException e) {
        System.out.println("i/o exception... bye!");
        System.exit(1);
    }
}
```

GuessNumber程序用一个ScanInput对象从终端读入数据并分析，ScanInput类的具体描述定义见附录A。

3.3 (重点) 平面中的直线段用`LineSegmentGeometry`对象表示。一条直线段有两个特性分别对应它的两个端点: `p0`特性对应一个端点, `p1`对应另一个。在一些情况下, 我们认为直线段是从`p0`端点(线段的起点)到`p1`端点(线段的终点)的有向线段。`LineSegmentGeometry`类有框架如下:

```
public class LineSegmentGeometry {  
  
    public LineSegmentGeometry(PointGeometry p0,  
                               PointGeometry p1)  
        throws NullPointerException  
    {  
        // EFFECTS: If p0 or p1 is null throws  
        // NullPointerException; else initializes this  
        // with endpoints equal to p0 and p1.  
  
        public LineSegmentGeometry(int x0, int y0,  
                                   int x1, int y1)  
  
        // EFFECTS: Initializes this with the endpoints  
        // (x0,y0) and (x1,y1).  
  
        public PointGeometry getP0()  
        // EFFECTS: Returns a copy of endpoint p0.  
  
        public void setP0(PointGeometry newP0)  
            throws NullPointerException  
        {  
            // MODIFIES: this  
            // EFFECTS: If newP0 is null throws  
            // NullPointerException; else updates the p0  
            // endpoint to a copy of newP0.  
  
        public PointGeometry getP1()  
        // EFFECTS: Returns a copy of endpoint p1.  
  
        public void setP1(PointGeometry newP1)  
            throws NullPointerException  
        {  
            // MODIFIES: this  
            // EFFECTS: If newP1 is null throws  
            // NullPointerException; else updates the p1  
            // endpoint to a copy of newP1.  
  
        public double length()  
        // EFFECTS: Returns the length of this segment.  
  
        public java.awt.Shape shape()  
        // EFFECTS: Returns the shape of this segment.  
  
        public void translate(int dx, int dy)  
        // MODIFIES: this  
        // EFFECTS: Translates this segment by dx and dy.  
    }
```

```

    // this_post.getp0().equals(getp0()+(dx,dy)),
    // this_post.getp1().equals(getp1()+(dx,dy)).

    public String toString()
    // EFFECTS: Returns the string "p0-p1".
}

```

下面的小程序使用LineSegmentGeometry类，该程序打印的串以斜体注释表示：

```

public class TryLineSegment {
    public static void main(String[] args) {

        LineSegmentGeometry a =
            new LineSegmentGeometry(1, 2, 3, 4);
        PointGeometry p = a.getp0();
        p.setX(88); // endpoint p0 of a not affected
        System.out.println("a: " + a); // a: (1,2)-(3,4)
        System.out.println("length: " + a.length());
        // length: 2.8284271247461903
        PointGeometry q = new PointGeometry(7, 8);
        a.setp0(q);
        q.setX(34); // endpoint p0 of a not affected
        System.out.println("a: " + a); // a: (7,8)-(3,4)
        a.translate(2, 3);
        System.out.println("a: " + a); // a: (9,11)-(5,7)
    }
}

```

注意，点p由表达式a.getp0()获得，p不能用于改变a的端点p0，这是因为表达式a.getp0()引用的对象是存储在线段a的p0域中的点的一个拷贝，而且这个对象并不是点a的状态的一部分。同样，直线a的状态也不会由于点q的改变而受影响，虽然点q被用来修改a的p0端点的值。通常情况下，直线段维护它自己的保护型端点（以PointGeometry对象的形式），不允许客户直接访问它的端点。下面给出shape方法的实现：

```

// method of LineSegmentGeometry class
public Shape shape() {
    return new Line2D.Float(p0.getX(), p0.getY(),
                           p1.getX(), p1.getY());
}

```

3.4（重点）一个属性（attribute）是一个名字-值对，名字是一个字符串，值是任意类型的对象。属性可以被用来表明一个对象的特性。例如，一个多边形可以有属性“颜色”：Color.red和“边数”：3，来表示它是一个红色的三角形。属性也可以用作字典的条目，每一个条目与一个带有值的唯一名字相联系。下面是一个Attribute类的框架结构：

```

public class Attribute {

    public Attribute(String name)
        throws NullPointerException

    // EFFECTS: If name is null throws
    // NullPointerException; else initializes
    // this to name:null.

    public Attribute(String name, Object value)
        throws NullPointerException
    // EFFECTS: If name is null throws

```

```

//    NullPointerException; else initializes
//    this to name:value.

public Object getValue()
    // EFFECTS: Returns this attribute's value.

public void setValue(Object newValue)
    // MODIFIES: this
    // EFFECTS: Changes this attribute's value
    //    to newValue.

public String name()
    // EFFECTS: Returns this attribute's name.

public boolean equals(Object obj)
    // EFFECTS: Returns true if obj is an Attribute
    //    and obj's name is equal to this attribute's
    //    name; else returns false.

public String toString()
    // EFFECTS: Returns the string "name:value".
}

```

注意, 值 (value) 是这个类的一个特性, 但是名字 (name) 不是 (没有方法可以改变一个属性的名字)。如果两个属性的名字相同, 则它们被认为是相等的, 无论它们的值是否相等。这里有一个小程序可以使用这个类:

```

public class TryAttribute {
    public static void main(String[] args) {
        Attribute apple1 = new Attribute("apple", "red");
        Attribute apple2 = new Attribute("apple");
        if (apple1.equals(apple2))
            System.out.println("this should print");
        System.out.println("apple1: " + apple1.name() +
            ":" + apple1.getValue()); // apple1: apple:red

        apple2.setValue("green");
        System.out.println("apple2: " + apple2);
        // apple2: apple:green
    }
}

```

实现Attribute类, 并且用TryAttribute或类似的测试程序测试它。

3.2.2 矩形

矩形是有四个直角和四条边的平面图形。这里假设矩形是标准定位 (standard orientation), 即它的边平行于x轴或y轴。假设矩形的长和宽都是非负整数值。这里定义的矩形位于坐标系的左上角。由于我们的坐标轴定位 (x轴向右增加, y轴向下增加), 所以矩形的位置 (position) 是x坐标和y坐标最小值所在角的那一点。矩形的宽度 (width) 是它的水平边的长度, 高度 (height) 是垂直边的长度 (见图3-2)。我们把矩形的位置、宽度和高度特性称为矩形的维数。

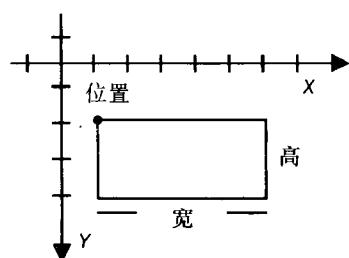


图3-2 有一个矩形的笛卡儿平面图

以下是我们矩形数据类型的类框架：

```

public class RectangleGeometry {

    public RectangleGeometry(int x, int y,
                           int width, int height)
        throws IllegalArgumentException
    // EFFECTS: If width or height is negative throws
    //   IllegalArgumentException; else initializes
    //   this to a rectangle at position (x,y) and of
    //   given width and height.

    public RectangleGeometry(PointGeometry pos,
                           int width, int height)
        throws NullPointerException,
               IllegalArgumentException
    // EFFECTS: If pos is null throws
    //   NullPointerException; else if width or height
    //   is negative throws IllegalArgumentException;
    //   else initializes this to a rectangle at
    //   position (pos.getX(),pos.getY()) and of
    //   given width and height.

    public RectangleGeometry(Range xRange, Range yRange)
        throws NullPointerException
    // EFFECTS: If xRange or yRange is null throws
    //   NullPointerException; else initializes this to
    //   a rectangle of specified x and y extents.

    public RectangleGeometry(RectangleGeometry r)
        throws NullPointerException
    // EFFECTS: If r is null throws
    //   NullPointerException; else initializes this to
    //   a rectangle with the same dimensions as r.

    public PointGeometry getPosition()
    // EFFECTS: Returns this rectangle's position.

    public void setPosition(PointGeometry p)
        throws NullPointerException
    // MODIFIES: this
    // EFFECTS: If p is null throws
    //   NullPointerException; else sets this
    //   rectangle's position to (p.getX(),p.getY()).

    public int getWidth()
    // EFFECTS: Returns this rectangle's width.

    public void setWidth(int newWidth)
        throws IllegalArgumentException
    // MODIFIES: this
    // EFFECTS: If newWidth is negative throws
    //   IllegalArgumentException; else sets this
    //   rectangle's width to newWidth.

    public int getHeight()
    // EFFECTS: Returns this rectangle's height.

    public void setHeight(int newHeight)
        throws IllegalArgumentException
    // MODIFIES: this
    // EFFECTS: If newHeight is negative throws
    //   IllegalArgumentException; else sets this
    
```

```

    // rectangle's height to newHeight.

public Range xRange()
    // EFFECTS: Returns the range spanned by
    // this rectangle's x coordinates.

public Range yRange()
    // EFFECTS: Returns the range spanned by
    // this rectangle's y coordinates.

public boolean contains(int x, int y)
    // EFFECTS: Returns true if the point (x,y) is
    // contained in this rectangle; else returns false.

public boolean contains(PointGeometry p)
    throws NullPointerException
    // EFFECTS: If p is null throws
    // NullPointerException; else returns true if p is
    // contained in this rectangle; else returns false.

public java.awt.Shape shape()
    // EFFECTS: Returns the shape of this rectangle.

public void translate(int dx, int dy)
    // MODIFIES: this
    // EFFECTS: Translates this rectangle by dx and dy:
    // this_post.getPosition() is equal to
    // this.getPosition()+(dx,dy).

public String toString()
    // EFFECTS: Returns the string
    // "Rectangle: (x,y),width,height".
}

```

在实现`RectangleGeometry`类之前，先看一下它的行为。我们先写一个名为`TryRectangle`的应用程序，它以四个描述矩形 r 的维数的参数（这些参数匹配`RectangleGeometry`的四个参数的构造器）被调用。`TryRectangle`程序输出矩形 r 的字符串描述符；然后程序给出提示，要求用户输入一个点的 x 坐标和 y 坐标；接下来程序输出用户输入的点，并且报告这些点在矩形内还是矩形外；然后程序进一步提示，用户再输入其他点，并且程序给出报告；用户输入点 $(0, 0)$ 结束程序。在下面的交互例子中，程序参数描述了一个矩形，它的左上角的坐标是 $(10, 10)$ ，宽是30，高是20：

```

> java TryRectangle 10 10 30 20
Rectangle: (10,10),30,20
? 20 15
(20,15): inside
? 40 30
(40,30): inside
? 41 30
(41,30): outside
? 10 16
(10,16): inside
? 0 0
(0,0): outside
>

```

应用程序通过`TryRectangle`类实现，静态方法首先分析参数并且用它们构造一个矩形 r 。然后，它打开一个输入流，并且反复给出提示“？””，从输入流中读下一个点的坐标，

构造该点，检测和报告该点是否包含在矩形r内。下面是实现：

```
public class TryRectangle {
    public static void main(String[] args) {
        if (args.length != 4) {
            String msg = "USAGE: java TryRectangle x y ";
            msg += "width height";
            System.out.println(msg);
            System.exit(1);
        }
        int x = Integer.parseInt(args[0]);
        int y = Integer.parseInt(args[1]);
        int width = Integer.parseInt(args[2]);
        int height = Integer.parseInt(args[3]);
        RectangleGeometry r =
            new RectangleGeometry(x, y, width, height);
        System.out.println(r);
        ScanInput in = new ScanInput();
        PointGeometry origin = new PointGeometry();
        while (true) {
            try {
                System.out.print("? ");
                x = in.readInt();
                y = in.readInt();
                PointGeometry p = new PointGeometry(x, y);
                System.out.print(p + ": ");
                if (r.contains(p)) System.out.println("inside");
                else System.out.println("outside");
                if (p.equals(origin)) break;
            } catch (NumberFormatException e) {
                System.out.println("please enter two numbers");
            } catch (IOException e) {
                System.out.println("i/o exception... bye!");
                System.exit(1);
            }
        }
    }
}
```

下面转向**RectangleGeometry**类的实现，该类定义了**Range**类型的两个域（见练习3.2）。

```
// fields of RectangleGeometry class
protected Range xRange, yRange;
```

这两个范围联合起来定义一个矩形：**xRange**域保存x坐标的范围（矩形的水平边跨度），**yRange**保存y坐标的范围（矩形的垂直边跨度）。例如，矩形

```
new Rectangle(new PointGeometry(1,2), 3, 6)
```

用下面两个范围表示：

```
xRange: new Range(1, 4)
yRange: new Range(2, 8)
```

注意，用来表示矩形存储结构方法有很多种，这里采用的仅仅是其中的一种。在练习中会探讨其他的可能性。

下面是类**RectangleGeometry**的四个构造器：

```
public RectangleGeometry(int x, int y,
                        int width, int height)
    throws IllegalArgumentException {
```

```

if ((width < 0) || (height < 0))
    throw new IllegalArgumentException();
xRange = new Range(x, x + width);
yRange = new Range(y, y + height);
}

public RectangleGeometry(PointGeometry pos,
    int width, int height)
throws NullPointerException,
    IllegalArgumentException {
    this(pos.getX(), pos.getY(), width, height);
}

public RectangleGeometry(Range xRange, Range yRange)
throws NullPointerException {
    this(xRange.getMin(), yRange.getMin(),
        xRange.length(), yRange.length());
}

public RectangleGeometry(RectangleGeometry r)
throws NullPointerException {
    this(r.getPosition(), r.getWidth(), r.getHeight());
}

```

第二个和第三个构造器直接调用第一个（四个参数）构造器，第四个间接调用第一个构造器。在这四个构造器中，只有第一个直接访问xRange和yRange实例域。

为了获得矩形的位置，我们提取xRange和yRange域的最小值：

```

// method of RectangleGeometry class
public PointGeometry getPosition() {
    return new PointGeometry(xRange.getMin(),
        yRange.getMin());
}

```

当更新矩形的位置时，需修改xRange和yRange的最小值，但矩形的宽和高不变：

```

// method of RectangleGeometry class
public void setPosition(PointGeometry p)
throws NullPointerException {
    xRange.setMinMax(p.getX(), p.getX() + getWidth());
    yRange.setMinMax(p.getY(), p.getY() + getHeight());
}

```

矩形的宽等于它的x范围的长度。下面是宽特性的获取者和设置者过程：

```

// methods of RectangleGeometry class
public int getWidth() {
    return xRange.length();
}

public void setWidth(int newWidth)
throws IllegalArgumentException {
    if (newWidth < 0) throw new IllegalArgumentException();
    xRange.setMax(xRange.getMin() + newWidth);
}

```

Shape方法返回这个矩形的形状对象：

```

// method of RectangleGeometry class
public java.awt.Shape shape() {
    PointGeometry p = getPosition();
    return new Rectangle2D.Float(p.getX(), p.getY(),

```

```

        getWidth(), getHeight());
}

```

两个`contains`方法报告输入点是否在这个矩形内。点 (x, y) 在矩形内，当且仅当 x 在矩形的 x 范围内和 y 在矩形的 y 范围内：

```

// methods of RectangleGeometry class
public boolean contains(int x, int y) {
    return xRange().contains(x) && yRange().contains(y);
}

public boolean contains(PointGeometry p)
    throws NullPointerException {
    return contains(p.getX(), p.getY());
}

```

练习

3.5（重点）完成类`RectangleGeometry`的实现，这需要你定义它的`getHeight`，`setHeight`，`xRange`，`yRange`，`translate`和`toString`方法。为了确保客户不能直接访问一个矩形的存储结构，`xRange`方法要返回由`xRange`引用的对象的一个拷贝（`yRange`方法中情况类似）。以下代码段给出`translate`和`toString`方法的行为：

```

RectangleGeometry r =
    new RectangleGeometry(new PointGeometry(2, 3), 4, 5);
System.out.println(r); // Rectangle: (2,3),4,5
r.translate(10, 20);
System.out.println(r); // Rectangle: (12,23),4,5

```

试着用另一种存储结构完成方法的实现（不用`xRange`和`yRange`域）。

3.3 封装

封装（encapsulation）是将相关软件元素组织到一起的过程。Java提供包（package）把一组相关类封装在一起。本章介绍一些更基本的基于对象的封装形式：对象封装了一组相关数据和方法。此外，组成对象的元素也可以是封装的，客户通过它们的接口和它们交互。

我们可以把一个对象看成是把一组数据和方法用包围层包起来，该对象中的一些元素又是用包围层包起来，这样就形成一种包围层嵌套。简单可以设想为：公有接口和私有实现都是对象中的元素。对象的公有接口包含在外包围层内；对象的私有实现部分在内包围层内（如图3-3所示）；而客户则在外包围层外。对象的外包围层是完全渗透的，因为客户可以通过它向其内部元素传递信息；而内包围层是半渗透性，因为它只接受特权客户的访问。所有客户都可以访问外包围层内的公有接口，只有特权客户才能访问内包围层内的元素。

封装的作用是把对象中不对外公开的元素封起来，称之为信息隐藏。对象和它的客户都受益于信息隐藏。因为客户不能访问对象的私有实现，也不能访问相应的私有接口，所以对象所有的交互要借助公有接口来完成，从而保证自身的状态的完整性。信息隐藏也有利于客户：客户依赖于对象固定的公有接口，所以对象内部实现的任何变化都不影

响客户的使用。

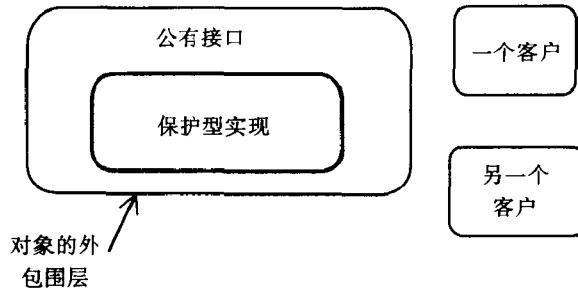


图3-3 对象内封装了一组接口

3.3.1 封装和类定义

类定义描述的是该类对象的方法和结构。类定义把对象的所有元素组织在一起，这样就简化了类的编写者、理解者和修改者的工作。类元素间关系十分紧密：类的操作是由类的方法实现的，而且这些方法可以访问和修改对象的实例域。只有在综合考虑全部元素的情况下，实现才可以被看成是一个整体。在类定义中的实现对编程者来说是比较方便的。

虽然如此，类定义在局部考虑元素时也有不足之处。这其中有两个问题：第一是其因为继承。当一个新类继承一个已有类时（经常在Java中出现），这个新子类继承了父类的“实现和接口”，那么这个新类的实现潜在地依赖于它的父类，而父类的实现又依赖于它的父类，以次类推，一直到原始类Object。这样一个新类的实现总与其他类（它的父型）有联系，这就导致所谓的yoyo问题，它使编程人员为了弄懂一个类，要在继承类链上前后反复地查看。为解决这个问题，集成开发环境中都有在浏览器中可以查看的继承层次结构，编程人员可以轻松地查看任何层次的类。

第二个问题是组成对象接口的元素和组成对象实现的元素在一个类定义中混合，尽管客户的编写者只需懂得接口来使用对象，但类定义包括接口和实现。看以下计数类，Up-Counter对象维护整数值，初始为0，以后每调用一次Inc方法就加1，该类还提供了value方法，该方法返回当前值，下面是该类的类定义：

```
public class UpCounter {
    private int value;

    public UpCounter() {
        // EFFECTS: Initializes this counter to zero.
        this.value = 0;
    }

    public void inc() {
        // MODIFIES: this
        // EFFECTS: Increments this counter by one.
        this.value++;
    }

    public int value() {
```

```

    // EFFECTS: Returns this counter's current value.
    return this.value;
}
}

```

客户的编写者只对UpCounter的接口感兴趣，但类定义却提供过多的信息，它提供了实现的细节，这就失去封装带来的信息隐藏的意义。参见图3-3，类定义提供了一个对象结构的视图和实现，无论读者是否想要或需要这样的信息。

Java的接口结构对从实现中分离接口是十分有用的。接口声明了一组操作而没有实现它们，它们的实现延迟到实现这个接口的类中。但是，给每个类都定义一个与之平行的接口是不切实际的。在实际应用中，要使用一个类还是常常要查看类定义的。

3.3.2 信息隐藏

信息隐藏用于隐藏不对外公开的对象元素，这类元素基本上包括所有数据域、用于同类或紧密相连类对象调用的非公有方法和所有方法的实现。

信息隐藏可以从一个问题引出：对象的实现对谁隐藏了它的信息？有两个答案：第一，对程序设计者隐藏。你也许会觉得奇怪，程序设计者定义类，当然会看到方法的实现。但是另一种经常发生的情况是：程序设计者一般依靠别人已设计好的类包，他们并不需关注包的源代码。但当程序设计者需要源代码时怎么办？即使这种情况，对象的实现也要从程序设计者的思想中隐藏，信息隐藏可以帮助我们区分不同水平的抽象。在编写代码或者设计开发系统中，当需要某些对象的服务时，对象的实现体——如何实现这些细节——不在考虑的范围内。不仅编译器强调对象的公有接口与保护型实现之间的区别，而且程序设计者思想中更要强调这种区别。

第二，对客户隐藏。客户有请求权限才能访问隐藏部分。例如，在RectangleGeometry类的xRange和yRange域定义为protected，在Java中这意味着只有RectangleGeometry的子类或和它在同一个包中的类才可访问它们。

Java提供四种不同的接口访问权限。用关键字表示它们分别是：public（公有）、protected（保护）、private（私有）和package（默认时为此权限）。定义类时，信息的公开和隐藏是在数据或方法成员前加上上面的任一关键字表示的。公有接口可被所有对象访问；私有接口只有同类的对象可以访问。下面是四种访问权限的解释：

- **public**: 所有对象都可以访问。
- **protected**: 属于同一包的类对象或任何子类的实例有权访问。
- **package**: 属于同一包的类对象可以访问。
- **private**: 仅属于同一类的对象可以访问。

上面权限列表从上到下，从公有到私有，访问权限的限制不断增加。接口变得不易访问却更实用。假设图3-3不是有两层，而是四层。最外层完全可渗透——可使任何消息经过——然而剩下的三层每一层都比上一层渗透性低，客户能访问哪一包围层内的服务是由它与对象的关系决定的，权限越高的客户可以进入更深层次，因而也可以享用更多服务。

举个简单例子，一个Java程序执行时输入一组整型参数，然后输出奇数数目和偶数数目。这个程序使用两个UpCounter类的实例（见3.3.1节）来计数：

```

public class EvenOddCount {
    public static void main(String[] args) {
        UpCounter even = new UpCounter();
        UpCounter odd = new UpCounter();
        for (int i = 0; i < args.length; i++) {
            int k = Integer.parseInt(args[i]);
            if (k%2 == 0) even.inc();
            else odd.inc();
        }
        System.out.println("number of evens: "+even.value());
        System.out.println("number of odds: "+odd.value());
    }
}

```

下面是程序的输入和输出示例：

```

> java EvenOddCount 0 2 4 6 9
number of evens: 4
number of odds: 1

```

在EvenOddCount程序中，main方法使用两个计数器（UpCounter类的实例），但它并没有直接访问计数器的隐藏实现。如果main程序要直接通过私有域来修改计数器的当前值编译时就会出现编译错误：

```
odd.value++;
```

正确的方法是通过设计一个操作来实现：

```
odd.inc();
```

信息隐藏防止客户不通过接口访问对象的信息，这样保证对象的状态不被客户胡乱修改，保持一致性。每个客户同对象的交互是通过该客户访问的接口的元素来进行的。

信息隐藏不仅保护服务对象而且保护了它的客户。因为客户直接依赖于它所访问的接口，和隐藏的实现没有任何联系，这样只要对象接口固定不变，客户就不会受对象实现的变化的影响。举例说明，假如我们不改变接口，修改UpCounter的实现部分，计数器当前值等于存储在value域中的值与最小整型值的差：

```

// class UpCounter: version 2
public class UpCounter {

    private int value;

    public UpCounter() {
        this.value = Integer.MIN_VALUE;
    }

    public void inc() {
        this.value++;
    }

    public int value() {
        return this.value - Integer.MIN_VALUE;
    }
}

```

UpCounter类第二版同原来的版本公有接口相同，由于EvenOddCount程序只依靠于这一接口而不依赖于实现，它的执行仍正确无误，它不受实现的变化的影响。相比之下，假如EvenOddCount程序直接访问计数器的value域，具体来说假设可以输入以下

语句来输出结果：

```
System.out.println("number of evens: " + even.value);
System.out.println("number of odds: " + odd.value);
```

这种情况下，程序的行为就会受UpCounter类实现的变化的影响。原先程序给出的输入现在的输出结果可能是错误的：

```
> java EvenOddCount 0 2 4 6 9
number of evens: -2147483644
number of odds: -2147483647
```

通过要求客户通过对象的公有接口使用对象，信息隐藏同时能帮助保证客户自身的正确性。

练习

3.6 RectangleGeometry类的域表示方法和它本身的特性不一致，如它的位置是一个点，但没有用一个保存点的结构表示，而是从xRange和yRange提取它的位置值。在本练习中，在保持它的说明不变的情况下，重新实现RectangleGeometry类以便它的存储结构与它和特性一致：

```
// fields of RectangleGeometry class
// (for this exercise)
PointGeometry pos;
int width, height;
```

例如，构造矩形，

```
new RectangleGeometry(new PointGeometry(1,2),3,6)
```

将被表示如下：

```
pos: (1,2)
width: 3
height: 6
```

RectangleGeometry类的最初实现的方法哪些必须修改？是否需要修改任何没有直接访问xRange和yRange域的方法？使用RetangleGeometry类的新版本来运行3.2.2节中的TryRectangle程序（新类和原来的类的接口完全一致，所以应该同样工作）。

3.4 Java图形基础

到现在我们已经学过几种图形对象，例如矩形和点，但还没有画过图形，我们将在后两节来补充这些内容。在这一节中，主要讲述Java中制作计算机图形的背景知识。然后在3.5节中，我们将编写一个Java程序，在窗口中绘画一个绿色矩形，这个程序虽然简单，但它将作为本书后续部分许多图形程序（包括你自己编写的程序）的模板。这个程序的图形的简单化将有助于我们集中注意力于所有这些图形程序的共性，而不被复杂图形的各种各样的要求细节所干扰。

3.4.1 Java 2D API绘图模型

绘图是将图形绘制到输出设备的过程。在Java 2D中，一个对象的绘图中心称作是绘图

环境 (rendering context) 或者图形环境 (graphics context)，它是 `java.awt.Graphics2D` 类的实例。绘图环境有三个目标：

- 表示绘图表面，它可以代表三种输出设备：屏幕、打印机或屏外缓冲器。
- 维护一组绘图属性的状态，包括绘画文本串的字体、画图填充用的画笔和轮廓线笔。
- 表示一组绘图图形对象的方法（如形状、文本和图像），把图形输出到绘画表面。

简短来说，Java的绘图分为四个步骤：

- 1) 为绘画表面获得一个 `Graphics2D` 对象。
- 2) 创建要输出的图形对象。
- 3) 按要求设置 `Graphics2D` 的属性。
- 4) 调用 `Graphics2D` 对象的绘图方法，所调用的绘图方法取决于要输出的图形对象和要达到的效果。

本节余下的部分将具体解释这四个步骤，同时会强调本书将使用的内容部分（例如，在屏幕上绘图，而不是输出到打印机和屏外缓冲器）。

3.4.2 获取绘图环境

带有标题条和边框的窗口称为框架 (frame)。在Java中，框架是显示在屏幕上的应用程序的主窗口。框架是一个容器，可以包容许多其他组件，例如按钮、文字域、标签和面板。这些组件被放置在框架标题条和边框围起来的矩形区域中，这个区域又称内容格 (content pane)。

框架中的一些组件是原子组件 (atomic component)，也就是说它们不能再包含其他组件，例如按钮、标签和文字域；其他组件则称为容器 (container)，它们可以包含其他组件，其中一个例子就是面板 (panel)。它是一个普通容器，常作为画布在其上进行图形绘制（我们所有的图形绘制都将在框架中的最高层面板中进行）。一般来说，一个框架包含一些高层组件，而这些组件中又有一些包含其他组件，而其他组件中又有一些包含组件等等，这样就形成组件的包含层次结构 (containment hierarchy)，这一层次结构的根是框架。从图形效果上看，框架是在最后面一层包含每个组件，每个组件出现在其容器组件的前面并被其父容器包含。

图3-4是一个程序的框架界面，这个程序用来维护名字-值对。框架的内容格中包括有三个面板，从上到下依次为：域面板、按钮面板和消息面板。域面板主要是用来让用户输入新名字-值对、删除和查找已存在的对等；按钮面板包含用户发命令的按钮；消息面板用来显示查找结果。图3-4的下面是一个组件层次树状结构图，树中的中间节点代表容器，叶子节点代表原子组件（文本、按钮和标记）。

框架在它的生存期中会经常刷新自己和其他组件。当框架被其他程序挡住，然后其他程序移开时，或者框架改变大小，或者框架第一次显示时，都会发生刷新。系统会自动检测到这些事件并提醒框架刷新自己。当框架内容有重大变化时，也要进行刷新。这些事件发生时，管理框架内容的程序会发 `repaint` 消息给框架，告诉它进行刷新。但 `repaint` 消息不会使框架马上刷新它自己，而是先计划刷新但需等下一次机会。

当框架刷新自己时，它会对所有组件进行刷新，就是说从根开始，重新绘制整个包含

层次结构。框架首先把自己内容格的背景刷新（灰色矩形），然后告诉它直接包含的组件进行刷新。原子组件会直接刷新自己，比如按钮先刷新背景颜色，然后是上面的文字；而容器组件却不同，每个容器（如面板）组件首先刷新它的背景和它的其他装饰如边框，然后告诉它的组件刷新自己。这样刷新过程从框架一直到包含层次结构的叶子节点。总的来说，每个容器先刷新自己，然后是其组件；每个原子组件直接刷新自己。

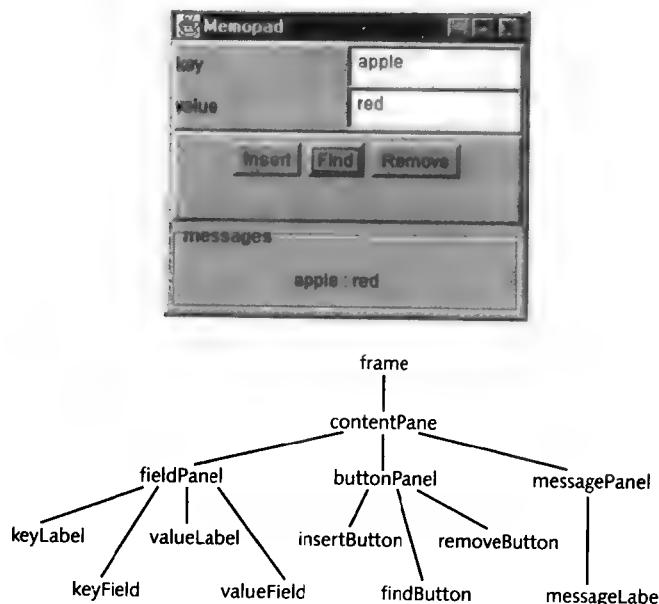


图3-4 Java应用程序和它的包含层次结构

当组件要刷新时，要执行它的`PaintComponent`方法，该方法的参数是一个绘图环境（`Graphics2D`类的实例），下面是`PaintComponent`方法签名：

```
// method of javax.swing.JComponent class
public void paintComponent(Graphics g)
```

尽管方法`JComponent.PaintComponent`的参数声明为`Graphics`类型，但实际传入的参数是`Graphics2D`，这是因为`Graphics2D`是`Graphics`的子型，这样做是为了与Java以前的版本兼容。

我们的图形程序都在屏幕上输出，特殊情况下图形画在框架内容格的面板上，这样面板就是绘图表面。一般图形程序都要覆盖定义`PaintComponent`方法，按自己的要求来绘图，`PaintComponent`方法可采用如下形式：

```
public void paintComponent(Graphics g) {
    super.paintComponent(g);
    Graphics2D g2 = (Graphics2D)g;
    // program-specific rendering using g2
    ...
}
```

把`g`强制转换成`Graphics2D`是为了保证方法可以访问绘图环境的全部功能。稍后会详细说明一般图形程序被细化和编写`paintComponent`方法的过程。

3.4.3 创建图形对象

三种图形对象可以用绘图环境输出：

- 形状 (shape)，一些几何图形类型，如矩形、椭圆、直线和曲线，每种形状实现 `Java.awt.Shape` 接口。
- 文本 (text)，可以按不同的字体、风格和颜色输出。
- 图像 (image)，为典型的 `java.awt.image.BufferedImage` 类的实例。

本书重点是在绘制形状上，也会用到一点文本。

图形对象的位置和大小是相对于坐标系的，又称为用户空间 (user space)。默认情况下坐标系的原点在绘图表面左上角， x 轴向右增加， y 轴向下增加。当在屏幕上输出时， x 和 y 轴以像素来度量。我们给出的形状图形总是相对于用户空间坐标系的（我们还将会看到用户空间变换，包括移动、缩放和旋转等）。

`java.awt.geom` 包中定义了很多实现 `java.awt.Shape` 接口的类，包括 `Rectangle2D.Float`，`Ellipse2D.Float` 和 `Line2D.Float` 等类。其实我们已使用过其中一些类，在 `PointGeometry.shape` 方法中曾创建并返回一个 `Ellipse2D.Float` 对象：

```
// method of PointGeometry class
public Shape shape() {
    return new Ellipse2D.Float(getX()-2, getY()-2, 4, 4);
}
```

这个过程产生一个中心在 (x, y) 直径为 4 的圆，其中前两个参数表示圆的外切矩形左上角的位置，后两个参数是矩形宽和高。因为 `Ellipse2D.Float` 类实现了 `Shape` 接口，所以 `shape` 方法返回类型 `Shape` 是合法的。

Java 2D 提供了许多形状类：椭圆、矩形、圆角矩形、弧线、直线、曲线、面积（面积提供了合并、交叉、异或的方法组合图形）。我们这里并不研究 Java 2D 支持的每个类的细节，而是随遇随讲。

3.4.4 设置绘图环境的属性

`Graphics2D` 对象封装了绘图用状态信息。它提供下面七个属性的设置和获得方法：

- `paint` 用来控制填充、绘画的像素颜色模式。
- `stroke` 控制图形轮廓线形状。
- `transform` 描述用户空间与绘图表面的相对关系。
- `front` 文本输出的字体。
- `composite` 描述绘图操作如何与存在的背景相混合。
- `clip` 剪切区域 (clip area)，它将绘图限制在一个区域内，绘图操作在这个区域外无效。
- `rendering hints` 提供控制绘图质量和速度的方法。

`Graphics2D` 类每个属性都提供获得者和设置者方法。例如，假设 `g2` 是 `Graphics2D` 对象，可以用下面语句设置 `g2` 的 `paint` 为绿色：

```
g2.setPaint(Color.green);
```

又如用下面语句获得绘图环境的当前 `stroke`：

```
Stroke stroke = g2.getStroke();
```

本书中主要使用paint、stroke和transform属性。

一个形状对象(Shape)可以有两种不同的绘制方法：内部填充(fill)和轮廓线绘制(stroke)。向Graphics2D对象发送fill消息进行内部填充；而绘制轮廓线时，要向Graphics2D对象发送draw消息，这两个方法都需要以形状对象为输入参数。

Graphics2D的paint属性用来确定填充和轮廓线的颜色模式，这个属性的值可以是任何实现java.awt.Paint接口的对象，最简单的是Color对象，表示某种固定颜色。Java还提供其他两个实现Paint的类，GradientPaint类提供由两种固定颜色形成的线性颜色，TexturePaint类可用图像颜色绘制。

属性值在改变前一直有效，例如，你可以以固定的蓝色填充一个矩形：

```
g2.setPaint(Color.blue);
g2.fill(new Rectangle2D.Float(10, 20, 80, 40));
```

随后用发送给g2的形状仍然为蓝色，直到g2的paint属性通过再次调用setPaint修改。

stroke属性确定绘制图形轮廓线的画笔的形状。Java有java.awt.BasicStroke类来表示stroke，你可确定stroke的宽度、如何打开stroke线段端点、两条线段如何交接、溅色的模式。BasicStroke对象的这些属性是创建时通过传递给构造器的参数控制的。它有五个不同构造器。Graphics2D的stroke属性设置是以BasicStroke对象为参数发送给它setStroke消息完成的。例如，以下代码画一条4像素宽的从(20, 30)到(50, 80)的直线：

```
g2.setStroke(new BasicStroke(4));
g2.draw(Line2D.Float(20, 30, 50, 80));
```

Graphics2D类的transform属性用来指定用户空间，也就是坐标系。默认情况下，用户空间等同于设备空间(device space)：坐标原点在绘图平面的左上角，x轴方向向右，y轴向下，坐标间距单位为像素(假设输出设备为屏幕)。使用这种默认用户空间就会产生一个问题：图形不能全部显示在屏幕上。例如，语句

```
g2.fill(new Ellipse2D.Float(-50, -50, 100, 100));
```

输出一个以原点为圆心，直径为100的圆。由于原点在绘图平面的左上角，所以该圆只能显示在绘图平面的左下角，事实上只有四分之一圆可以看到。为解决这个问题，我们首先把用户空间的原点转移到和绘图平面中心相一致。假设绘图平面的长和高由变量width和height决定，则可用下面的代码段移动用户空间：

```
g2.translate(width/2, height/2); // translate g2
g2.fill(new Ellipse2D.Float(-50, -50, 100, 100));
```

现在再画圆时，它就出现在绘图平面的中间了。在没有再次移动用户空间前，g2会一直使用这个用户空间。

我们即将要介绍的许多图形程序都要使用paintComponent方法，它的定义一般是从如下形式开始的：

```
public void paintComponent(Graphics g) {
    super.paintComponent(g);
    Graphics2D g2 = (Graphics2D)g;
    // translate the origin of g2's user space to
```

```

    // the frame's center
    Dimension d = getFrame().getContentSize();
    g2.translate((int)(d.width/2), (int)(d.height/2));
    // program-specific rendering using g2
    ...
}

```

这样在图形输出前，就把g2用户空间的原点移到绘图平面中心。在图3-5中，黑色坐标系是设备空间，灰色坐标系是移动过的用户空间。

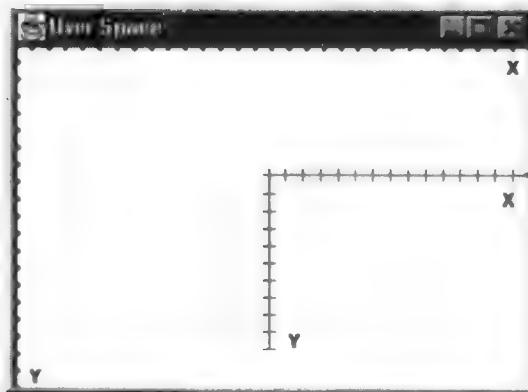


图3-5 黑色坐标为设备空间，灰色坐标为移动过的用户空间

`translate`方法只是`Graphics2D`提供的坐标变换多种方法中的一种。`Graphics2D`类还提供`rotation`、`scale`和`shear`方法。它们与`translate`组织在一起称为基本变换 (elementary transform)。`rotation`用来围绕任意点旋转用户空间，`scale`用来通过x和y轴变化来放大或缩小用户空间，`shear`是使用户空间倾斜，包括按比例移动 (例如会使矩形变成平行四边形)。`Graphics2D`类还提供`getTransform`和`setTransform`方法来对用户空间直接进行操作，它们对一个称为变换 (transform) 的对象 (`AffineTransform`类的一个实例) 进行这些操作。总体上说，除了要把用户空间移到绘图平面中心外，和绘图环境相关的用户空间是可以很好地完成它的任务的。这部分将在第6章继续讲述。

3.4.5 绘图

前面已经提到过，`Graphics2D`对象可以画出三种图形对象：形状、文本和图像。本书重点是形状，它是实现`java.awt.Shape`接口的对象。形状有两种画法：内部填充 (使用`fill`方法) 和轮廓线绘制 (使用`draw`方法)，两个方法都以`Shape`对象为参数：

```

// methods of Graphics2D class
public void fill(Shape s) throws NullPointerException
    // EFFECTS: If s is null throws NullPointerException;
    // else fills the interior of s based on this
    // rendering context's attributes.
public void draw(Shape s) throws NullPointerException
    // EFFECTS: If s is null throws NullPointerException;
    // else strokes the outline of s based on this
    // rendering context's attributes.

```

例如，下面代码段用绿色填充直径为100的圆：

```
g2.setPaint(Color.green);
```

```
Shape circle = new Ellipse2D.Float(50, 50, 100, 100);
g2.fill(circle);
```

圆出现在与绘图环境g2相关的绘图平面上。

要画出形状的轮廓线，要发送以形状作为参数的一个draw消息给Graphics2D对象。例如，下面代码以红色、4像素宽画笔，画出左上角点为(10, 20)、宽80、高40的矩形的轮廓线：

```
g2.setPaint(Color.red);
g2.setStroke(new BasicStroke(4));
Shape rectangle = new Rectangle2D.Float(10, 20, 80, 40);
g2.draw(rectangle);
```

矩形中间没加填充，只画出轮廓线。

当然可以同时对同一形状进行填充和轮廓线绘制。这样设置绘图环境属性语句与完成实际绘画的语句交叉出现，下面的代码假设Graphics2D对象g2已建立：

```
// construct a graphical object
Shape shape = new Rectangle2D.Float(50, 50, 150, 100);

// set rendering context's attributes and fill
g2.setPaint(Color.lightGray);
g2.fill(shape);

// set rendering context's attributes and draw outline
g2.setPaint(Color.darkGray);
g2.setStroke(new BasicStroke(8));
g2.draw(shape);
```

图3-6显示该段代码产生的图形结果，其中坐标轴间距单位50。

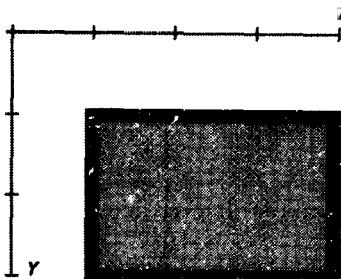


图3-6 带填充和轮廓线的矩形

3.5 Java图形程序实例

在这一节中，我们将设计一个在黑色的框架背景上输出绿色矩形的程序。在这个过程中，将设计一个将来所有图形程序都使用的程序模板。

3.5.1 画矩形

在第7章之前，所有的图形程序中框架的内容格中只包含一个组件——面板。这个面板占据框架内容格的全部空间，并作为画布在其上进行图形绘制。在开发图形程序时，要定义一个ApplicationPanel类的子类。ApplicationPanel类（定义请见附录B）是在继承其父类javax.swing.JPanel的基础上还加了一些其他的行为的面板。作为

ApplicationPanel类的子类，该类同时也是一个面板，类中的域和方法用来产生图形。

面板包含一个ApplicationFrame对象，它是框架中的一种。ApplicationFrame类（定义请见附录B）在标准框架的基础上增加一些其他行为，特别是它也可以按标准方法关闭（如点击框架的关闭按钮）或出现在屏幕中央。

我们将要设计的画绿色矩形的程序名为PaintOneRectangle，该程序通过执行下面的命名行来调用：

```
> java PaintOneRectangle x y width height
```

该程序建立一个新框架，在这个框架上输出一个绿色矩形，其位置在(x, y)，宽width，高height。程序整体如下：

```
public class PaintOneRectangle extends ApplicationPanel {

    public PaintOneRectangle() {
        setBackground(Color.black);
    }

    public static void main(String[] args) {
        parseArgs(args);
        ApplicationPanel panel = new PaintOneRectangle();
        ApplicationFrame frame =
            new ApplicationFrame("A Green Rectangle");
        frame.setPanel(panel);
        panel.makeContent();
        frame.show();
    }

    // static fields storing parsed program arguments
    protected static int x, y, width, height;

    // parses the program arguments
    public static void parseArgs (String[] args) {
        if (args.length != 4) {
            String s = "USAGE: java PaintOneRectangle x y w h";
            System.out.println(s);
            System.exit(0);
        }
        x = Integer.parseInt(args[0]);
        y = Integer.parseInt(args[1]);
        width = Integer.parseInt(args[2]);
        height = Integer.parseInt(args[3]);
    }

    // instance field representing
    // the contents of this panel
    protected RectangleGeometry rectangle;

    // creates the graphical contents of this panel
    public void makeContent() {
        rectangle =
            new RectangleGeometry(x, y, width, height);
    }

    // paints the contents of this panel
    public void paintComponent(Graphics g) {
        super.paintComponent(g);
        Graphics2D g2 = (Graphics2D)g;
        g2.setRenderingHint(RenderingHints.KEY_ANTIALIASING,
```

```

        RenderingHints.VALUE_ANTIALIAS_ON);
        g2.setPaint(Color.green);
        g2.fill(rectangle.shape());
    }
}

```

下面一个接一个地讨论PaintOneRectangle类中的元素。PaintOneRectangle构造器中设置面板背景为黑色。因为组件默认的背景颜色是灰色，所以如果不定义这个构造器，该程序将产生一个背景为灰色的绿色矩形。

PaintOneRectangle程序执行从静态main方法开始，它被调用时带有四个参数，main过程完成如下动作：

- 1) 调用parseArgs过程分析程序参数。
- 2) 创建新的PaintOneRectangle对象（保存在局部变量panel中）。
- 3) 创建新的框架（保存在局部变量frame中）。
- 4) 增加面板对象到框架中。
- 5) 向面板panel发送makeContent消息，创建图形。
- 6) 向框架frame发送show消息，使框架可视化。

parseArgs方法把程序参数转换为程序可用的格式，这里是把程序参数x, y, width和height都转换成整数，并保存在同名静态变量中。

窗口中的绿色矩形由程序的PaintOneRectangle类的实例管理（由main创建，到这一实例的引用保存在局部变量panel中）。这是一个面板实例，它定义了两个新的实例方法：makeContent方法创建面板图形；paintComponent方法绘制面板图形。详细地说，makeContent创建图形内容，这里是由RectangleGeometry对象表示的绿色矩形，对这个矩形的引用被保存在实例域rectangle中。paintComponent方法在面板中绘制绿色的矩形。为了做到这一点，PaintComponent先给父类一个绘图的机会；然后将参数g强制转换成Graphics2D对象，命名为g2；接着调用g2.setRenderingHint进行一些保证图形质量的设置；然后该方法发送给g2一个setPaint消息来设置填充色为绿色；最后发送给g2一个fill消息并把矩形的形状当作参数传入。

3.5.2 图形程序模板

设计PaintOneRectangle程序的主要目的是为其他图形程序建立一个基本模板。基于这种考虑，我们将把PaintOneRectangle程序所用的模板分离出来，下面是这个程序所采用的形式：

```

public class MyGraphicsProgram extends ApplicationPanel {
    // constructor
    public MyGraphicsProgram() {
        ...
    }

    public static void main(String[] args) {
        parseArgs(args);
        ApplicationPanel panel = new MyGraphicsProgram();
        ApplicationFrame frame =
            new ApplicationFrame("My Program");
        frame.setPanel(panel);
        panel.makeContent();
    }
}

```

```

        frame.show();
    }

    // static fields storing parsed program arguments
    ...

    // parses program arguments
    public static void parseArgs(String[] args) {
        ...
    }

    // primary storage structure
    // instance fields representing the contents of this
    // panel go here:
    ...

    // make the contents of this panel
    public void makeContent() {
        ...
    }

    // paint the contents of this panel
    public void paintComponent(Graphics g) {
        super.paintComponent(g);
        Graphics2D g2 = (Graphics2D)g;
        g2.setRenderingHint(RenderingHints.KEY_ANTIALIASING,
            RenderingHints.VALUE_ANTIALIAS_ON);
        ...
    }
}

```

基于这个模板开发时，用你的程序名称替换三个MyGraphicsProgram，而且还要定义你的程序中特定的域和完成方法的实现，这就要求你用你的代码替换现在为省略号（…）的地方。

Java程序中，对编译器来说，静态的main方法是必须定义的。模板中其余方法在父类ApplicationPanel中什么也没有做，所以当你要在你的程序中实现这些方法时，就要覆盖这些什么也没做的方法。实际应用时，有时并不需要定义你的程序的构造器；并且只有你的程序执行时有参数，才须定义parseArgs过程；但如果程序要创建图形（不论何种图形），就有必要定义一个存储结构，并要实现使用这些存储结构的makeContent和paintComponent方法。如果不这样做，程序只会建立一个空白窗口，这样就太不生动了，连我们前面创建的绿色矩形程序都不如。

练习

3.7 修改PaintOneRectangle程序，使它执行时可以带有七个参数，最后面三个用来表示填充矩形的颜色：

```
> java PaintOneRectangle x y width height [red green blue]
```

后面三个参数为可选，如果你的程序以四个参数调用而不是七个参数矩阵填充色为绿色。

3.8 再修改上一练习中的程序，使程序执行时可以带八个参数，其中第八个参数为整型数：

```
> java PaintOneRectangle x y width height [red green blue [s-width]]
```

矩形用输入的颜色填充，程序的轮廓线用宽度为 `s-width` 的白色画笔画出。如果 `s-width` 没有输入，则程序功能和前一个练习一样（填充矩形而不画出轮廓）。

小结

数据抽象将一个数据值表示成一组数据和一组操作，这些操作构成这些数据的公有接口，客户正是通过公有接口访问数据。信息隐藏屏蔽了对象的内部结构和操作的实现，这样客户不能随意修改对象的状态，而客户也不依赖于对象的具体实现细节，从而只与对象的固定公有接口交互。对象具体实现细节的修改并不影响客户按原来的方式正常工作。客户的设计者也只需熟悉对象的公有接口，不用理解该对象是如何工作的。一个对象的行为可以用一种表示方法描述：每个方法分别用需求子句、修改子句、作用子句说明方法的需求和功能过程（见第2章）。

封装是将相关软件元素组织到一起的过程。包封装相关的类；对象封装相关的数据、方法。封装是可以嵌套的：对象的组成元素可以是由其他元素封装形成的。对象可以分成公有部分（公有接口）和其他访问权限逐渐减小能力增强的部分。最受限制的部分（对象的私有接口）只允许同一类型对象访问，但它是能力最强的部分。信息隐藏是把对象中实现的元素对外隔离。

抽象和信息隐藏是互补的概念。它们侧重点不同，抽象强调客户使用对象需要知道什么，而信息隐藏则强调客户不要知道对象的某些部分，以免误用。它们二者结合起来使用就可以利用对象的服务并能隐藏保护它的内部实现细节。

第4章 组合

面向对象程序设计的一个重要优点是支持软件元素的重用。在这一章中我们将学习一种重用的基本机制——组合（composition）。组合用来建立包含其他对象的新对象，从而实现了（被包含）对象的重用。包含其他对象的对象被称为组合体（composite），它包含的对象是它的组件（component）。组合体和它的组件之间是一种整体-部分的关系：组合体是整体，而其组件是部分。在自然界和人造世界中有大量关于组合的例子：一台个人电脑是由中央处理器、内存、辅助电子设备和外设组成的；一本书是由许多章节构成的；一株植物是由根、茎、叶组成的。

4.1节概述了组合和聚集，聚集是类之间的一种关联关系。4.2节设计了一些生成各种随机数的组合类，这些类在后面被用来生成一些有趣的图片。4.3节设计了一个用来描述平面上折线的组合类，折线类的组件是由顶点组成的集合，这个集合的大小事先并不清楚。4.4节介绍了表达一致性约束（representation invariant）的概念，它是用一些用来描述使对象保持一致状态的条件。利用表达一致性约束将设计两个类，一个用来表示椭圆，一个用来表示有理数。4.5节介绍一个交互式图形程序的模板，使用户可以控制程序的运行。

4.1 组合和聚集

回忆第1章可以知道：如果两个类的实例是相关的，那么在两个类中存在一种关联。组合和聚集就是两种类型的关联，它们都是表示整体-部分的关系模型。在这两种关联下，表示整体的类对象包含表示部分的类对象。但组合和聚集是有区别的，区别在于它们表示的整体-部分的情况不同。

组合用来采用分层结构创建复杂对象。一个组合对象包含一个或多个较简单对象，依次地，那些较简单的对象包含更简单的对象，最后直到最简单（原子组件）对象为止。举例来说，考虑一株植物的结构，每一株植物的组件（根、茎、叶）合起来是一个组合；考虑其中一个组件——叶子，一片叶子由叶身、叶柄和将叶子连到树枝上的叶根组成，依次下去，叶身包含三种组织（表皮、叶肉、叶脉）。组成各种不同组织的细胞是由叶绿体和原子组成的。而叶绿体是由叶绿素和酶组成，酶是由DNA、线粒体和其他原子组织组成的。一株植物最高层的组件——根和茎，也可以同样分析。

一个组合对象完全拥有它的组件，部分与整体共存，如整体不存在了，部分也会随之消失。对象B是对象A的一个组件，仅当满足下面两个条件：（1）B属于A但不能同时属于除A之外的其他对象；（2）对象B由对象A创建并释放。这种强所有关系在Java程序设计中表现为：对象B是对象A的一个组件，如果同时满足下面两条件：

- 1) B只能由A访问，除了A其他对象不能访问对象B。
- 2) B的生存期由A控制。

我们把这些条件应用到3.2节的RectangleGeometry类上。这个类的每个实例都包

含 `xRange` 和 `yRange` 域。它们两个都是类 `Range` 对象的引用，是矩形对象的组件。条件1被满足，因为 `xRange` 和 `yRange` 不能被除了矩形之外的其他对象引用，甚至当一个客户调用 `xRange` 方法时，返回值是 `xRange` 引用的 `Range` 对象的一个拷贝，而不是对象自身；条件2被满足，因为 `xRange` 和 `yRange` 引用的两个 `Range` 对象都是由矩形对象构造器创建的，在矩形对象被释放时才释放，即它们的生存期由拥有它们的矩形对象控制。所以说每一个 `RectangleGeometry` 对象是一个组合体，`xRange` 和 `yRange` 域引用的两个 `Range` 对象是它的组件。

图4-1展示了 `Range` 和 `RectangleGeometry` 类的UML类图。实心菱形代表 `RectangleGeometry` 类是由 `Range` 对象（多重性值2表示一个矩形由两个 `Range` 对象组成）组成的。类图是以一条连接两个类的直线加上在组合体类端的一个实心菱形来表示组合的。

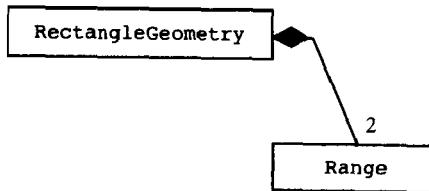


图4-1 一个矩形对象由两个 `Range` 对象组成的类图

聚集 (aggregation) 是另一种表示类之间整体 - 部分关系的关联。在聚集中，一个对象包含其他对象作为自己的一部分，但是不用满足前面的条件1和条件2，即组件可以同时属于多个组合体，不被某一个组合体所单独占有。举个例子，一支运动队由队员组成，但有些队员或许还属于其他运动队，当然队员的生命也不会由运动队控制（尽管有时会有非自然的惨剧发生）。因此一支运动队就是一个聚集整体，它的队员就是组成部分。类的聚集是由一条连接两个类的直线，加上在组合体类端的一个空心菱形来表示（见图4-2）。

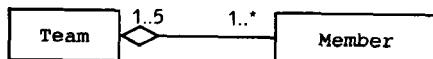


图4-2 表明一个运动队是由一个或多个成员组成的聚集，而一个成员
可以同时属于一个到五个运动队的类图

聚集常常用在组合不适用的情况下，如当部分被不止一个整体共享时，就要考虑用聚集而不用组合。然而虽然聚集是一种关联，但聚集的语义在UML中却没有详细的定义，所以尽管UML提供了表示聚集的符号，它的精确定义却可能随不同的项目而不同。这与组合形成鲜明的对比，UML详细描述了组合的含义。本章将着重讲述组合。

4.2 随机数生成器

在这一节我们将设计几个用来生成各种随机数的类：整数、点、矩形、颜色。这些不同的随机数生成类将由组合联系起来，形成一个关联网，其类图如图4-3所示。

在此之前，我们看一下术语随机的意义。为了简化讨论，我们假定问题中的值就是数。一个数被随机选中意味着它虽然被选中，但它和其他数被选中的几率是相同的。我们就说

一个随机数 (random number) 跟其他数可能出现的条件是相等的, 从技术上说, 这样的数被称为统一随机数 (uniform random number), 意思是说它服从统一分配。本书始终将随机视为统一随机。

随机数的选择范围是有限的或至少是有边界的。例如, 我们可以从限定范围0和1之间挑出随机小数; 或者从0到1000之间选出随机整数, 我们用标识[1..1000]表示限定的范围。

我们经常会用到随机数的序列。例如, 一个程序可以得到一个在各个方向随机出现的点, 随机数可用来决定取哪个方向, 如果程序频繁得到这样的随机点, 它就需要按次序将随机数排列。(这样的程序实现了所谓的称为随机算法 (randomized algorithm))。随机数序列在计算机图形中也可用来创建一组有用的对象, 例如给定矩形中的一组随机点、一组随机颜色或一组随机三角形组成一个网状图形。在计算机图形中, 随机产生的对象能产生非常有趣的图形效果, 这是因为我们想象中产生的图形合乎规则, 而实际上经常会有意想不到的惊奇的事情发生。

随机数序列由称为随机数生成器 (random numbers generator) 的程序产生。我们将使用的随机数生成器是确定的 (deterministic), 也就是说它们通过一个明确定义且可重现的步骤产生随机数序列。这些由确定随机数生成器产生的数被称为伪随机数 (pseudorandom number)。因为伪随机数序列和真随机数序列有很多相同的特性, 它们几乎可以等价使用。使用确定随机数生成器的一个优点是可以用一个称为种子 (seed) 的数初始化, 这个种子将随机数生成器置于一种特殊的状态, 从而保证它产生一个明确的随机数序列, 直到下一次重新播种。不同的种子一般会产生不同的随机数序列, 或者至少是在一个相同的很长随机序列里有不同开始点。相同的种子通常会产生相同的随机序列。只要知道随机数生成器和种子, 就有可能重复产生相同的随机数序列。

4.2.1 Java的Random类

Java提供了随机数生成类java.util.Random。下面这个类框架只包含了我们在本书中将用到的方法:

```
public class java.util.Random {

    public Random()
        // EFFECTS: Seeds this new object with the system time.

    public Random(long seed)
        // EFFECTS: Seeds this new object with the value seed.

    public void setSeed(long seed)
        // MODIFIES: this
        // EFFECTS: Reseeds this with seed: places this in
        // the same state as the object new Random(seed).

    public int nextInt()
        // MODIFIES: this
        // EFFECTS: Returns the next random int, and changes
        // the state of this to reflect the return
        // of this new value.
}
```

这个无参数构造器以当前时间为种子建立了新的Random对象, 构造器可以这样定义:

```
public Random() {
```

```

    this(System.currentTimeMillis());
}

```

无参数构造器调用一个参数的构造器，参数为种子。

请注意 `setSeed` 操作的后置条件。用传入的种子初始化已有的 `Random` 对象，将对象状态设置成与使用同一种子构造的新的 `Random` 对象的状态一样：一直到重新播种前，它们都会生成相同的随机数序列。举例来说，下面这一段代码将永远不会显示 “this should not print”：

```

long anyPositiveInteger = 10000;
long anySeed = 1234597;
Random rnd1 = new Random(anySeed);
Random rnd2 = new Random();
rnd2.setSeed(anySeed);
for (int i = 0; i < anyPositiveInteger; i++)
    if (rnd1.nextInt() != rnd2.nextInt())
        System.out.println("this should not print");

```

下面一段程序使用了 `Random` 类。它模仿重复的掷硬币动作，然后记住随机的正面和反面的次数，输出结果。如果程序带可选参数 `n` 被调用，则掷硬币 `n` 次，否则为 1000 次：

```

public class CoinFlip {
    public static final int HEAD = 0, TAIL = 1;

    public static void main(String[] args) {
        long nbrFlips = 1000;
        if (args.length > 1) {
            System.out.println("USAGE: java CoinFlip [nbrFlips]");
            System.exit(1);
        } else if (args.length == 1)
            nbrFlips = Long.parseLong(args[0]);
        long nbrHeads = 0, nbrTails = 0;
        java.util.Random rnd = new java.util.Random();
        for (long i = 0; i < nbrFlips; i++) {
            int side = Math.abs(rnd.nextInt()) % 2;
            if (side == HEAD) ++nbrHeads;
            else ++nbrTails;
        }
        String msg1 = "Number of heads = " + nbrHeads;
        msg1 += "(" + ((float)nbrHeads/nbrFlips * 100) + "%)\n";
        String msg2 = "Number of tails = " + nbrTails;
        msg2 += "(" + ((float)nbrTails/nbrFlips * 100) + "%)";
        System.out.println(msg1 + msg2);
    }
}

```

在本章余下部分将设计各种不同类型随机数的类：整数、点、矩形、颜色。不论这些类实例产生的是随机对象还是像整型这样基本的数据类型，都称它们为随机数生成器。这里的每个随机数生成器在很大程度上和 Java 的 `Random` 类形式相似，除了一些小的变化外，每个类都提供：

- 无参数构造器使用的种子源于系统，通常是系统时间。
- 带有一个参数的构造器，种子由客户提供。
- 一个由新种子初始化随机数生成器的方法。
- 一个或多个获取下一个随机值的方法。

因为有相当多的类将在本节中讨论，所以这里先给出其类图（图 4-3）。每当有新类介

绍时, 请参考这个类图。后面练习会要求你定义几个和此类图中类相关的类, 你也许想把自己定义的类加入到这个图中。

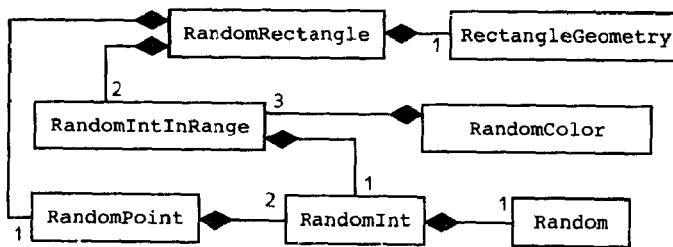


图4-3 一些由组合定义的随机数生成器

练习

- 4.1 大数定律表明: 随着掷硬币次数 n 增加, 正反两面的概率总是各占 50%。通过运行 CoinFlip 程序, 不断增加 n 的值测试这一点是否正确。
- 4.2 编写一个 DieRoll 程序, 它模拟掷一个六面骰子 (上面有 1 到 6 个数字, 如同 CoinFlip 掷一个硬币) n 次, 然后记录 1 到 6 出现的次数。这里 n 由可选程序参数给出, 或者如果没有参数输入, n 默认等于 1000。
[提示: 用一个长度为 6 的整型数组存储每个数字出现的次数。]
- 4.3 编写一个 DiceRoll 程序, 模拟同时掷两个骰子, 记录每次的出现的两个数字的和 (该数字应有 2 到 12 之间)。掷 n 次之后, 程序输出每个数字 (2 到 12) 出现的次数。 n 是一个可选程序参数, 如果不输入则默认为 1000。

4.2.2 随机整数

由 `Random.nextInt` 方法返回的随机整数在 Java 的 `int` 类型的范围内: -2^{31} 到 $2^{31} - 1$ 。有时你会希望从某个限定范围内选择随机数, 因而我们将要讨论的 `RandomInt` 类允许每次产生随机数时可改变选择范围。

像 Java 的 `Random` 类一样, `RandomInt` 类定义了一个 `nextInt` 方法用来获取下一个随机数, 但在该类中还对这个方法进行了重载, 被调用的特定的参数列表确定下一个随机数的选择范围。下面是这个类的框架:

```

public class RandomInt {

    public RandomInt()
        // EFFECTS: Seeds this new object with
        // the system time.

    public RandomInt(long seed)
        // EFFECTS: Seeds this new object with seed.

    public void setSeed(long seed)
        // MODIFIES: this
        // EFFECTS: Reseeds this with seed: places this
        // in the same state as new RandomInt(seed).

    public int nextUnsigned()
  
```

```

// MODIFIES: this
// EFFECTS: Returns the next nonnegative random
//   int, and updates the state of this to reflect
//   the return of this new value.

public int nextInt()
    // MODIFIES: this
    // EFFECTS: Returns the next random int and
    //   updates the state of this.

public int nextInt(int n)
    throws IllegalArgumentException
    // MODIFIES: this
    // EFFECTS: If n <= 0 throws IllegalArgumentException;
    //   else returns the next random int in the range
    //   [0..n-1] and updates the state of this.

public int nextInt(int m, int n)
    throws IllegalArgumentException
    // MODIFIES: this
    // EFFECTS: If n < m throws IllegalArgumentException;
    //   else returns the next random int in the range
    //   [m..n] and updates the state of this.

public int nextInt(Range rng)
    throws NullPointerException
    // MODIFIES: this
    // EFFECTS: If rng is null throws
    //   NullPointerException; else returns the next
    //   random int in rng and updates the state of this.

}

RandomInt将作为一个组合类实现，它的惟一组件是Random对象，存储在rnd实例域中：
```

```
// field of RandomInt class
protected Random rnd;
```

RandomInt有两个构造器：一个不带参数，取当前时间为种子；另一个有一个参数，给定种子：

```
public RandomInt() {
    rnd = new Random();
}

public RandomInt(long seed) {
    rnd = new Random(seed);
}
```

RandomInt对象可以使用setSeed方法重置种子：

```
// method of RandomInt class
public void setSeed(long seed) {
    rnd.setSeed(seed);
}
```

nextInt方法返回一个随机非负整数：

```
// method of RandomInt class
public int nextInt() {
    int i = rnd.nextInt();
    if (i < 0) i = -(i + 1);
    return i;
}
```

`nextInt`方法被重载，这一函数的无参数版本直接向`rnd`域引用的`Random`对象转发一个请求消息：

```
// method of RandomInt class
public int nextInt() {
    return rnd.nextInt();
}
```

组合对象把消息直接转发给它的一个组件的技术称为是委托 (delegation)。在上面刚刚给出的`nextInt`方法的无参数版本中，`nextInt`消息被委托给组件`rnd`。上面的`setSeed`方法的实现也用到了委托。

当`nextInt`方法带单个整型参数`n`调用时，它返回一个在 $[0..n-1]$ 之间的随机数：

```
// method of RandomInt class
public int nextInt(int n)
    throws IllegalArgumentException {
    if (n <= 0) throw new IllegalArgumentException();
    int i = nextUnsigned();
    if (i >= (n * (Integer.MAX_VALUE / n)))
        return nextInt(n);
    else
        return i % n;
}
```

当`nextInt`方法带整数参数`m`和`n`调用时，它返回一个在 $[m..n]$ 范围内的随机数。这一方法的实现是先在 $[0..n-m]$ 之间生成随机数，然后加上偏移量`m`：

```
// method of RandomInt class
public int nextInt(int m, int n)
    throws IllegalArgumentException {
    int rangeSize = n - m + 1; // nbr of integers in [m..n]
    return nextInt(rangeSize) + m;
}
```

最后一个`nextInt`方法的参数为一个范围，生成的随机整数要在这个范围中。下面是其实现：

```
// method of RandomInt class
public int nextInt(Range rng)
    throws NullPointerException {
    return nextInt(rng.getMin(), rng.getMax());
}
```

练习

4.4 考虑下面的`RandomInt`的`nextUnsigned`方法的另一个版本，它返回从`rnd`获取的整数的绝对值：

```
// method of RandomInt class: version2 (faulty)
public int nextUnsigned() {
    int i = rnd.nextInt();
    return (i >= 0) ? i : -i;
}
```

思考一下为什么`nextUnsigned`的版本2有错？

[提示：有两个问题。第一，找一个负整数，它的值取反不是一个正整数；第二，考虑是否每一个在 $[0..2^{31}-1]$ 范围内的整数能被等几率选择。]

4.5 这是 `nextUnsigned` 方法的第3个版本。它的思想是重复从 `rnd` 得到随机整数，直到获得一个非负值为止，然后返回：

```
// method of RandomInt class: version 3
public int nextUnsigned() {
    int i = rnd.nextInt();
    while (i < 0)
        i = rnd.nextInt();
    return i;
}
```

比较 `nextUnsigned` 方法的版本3和版本1的实现，版本3有版本1效率高吗？如果没有，用一两句话描述这样做效率低多少。

4.6 下面是 `randomInt` 的单参数 `nextInt` 方法的另一种实现：

```
// method of RandomInt class: version 2
public int nextInt(int n)
    throws IllegalArgumentException {
    if (n <= 0)
        throw new IllegalArgumentException();
    return nextUnsigned() % n;
}
```

尽管这个版本比原来的简单，但它是相对更好的，为什么？

4.7 用 `RandomInt` 对象代替 `Random` 对象，修改 4.2.1 节中的 `CoinFlip` 程序的实现。

4.2.3 固定范围内的随机整数

使用我们刚定义的 `RandomInt` 类，你能在每次使用 `nextInt` 方法时改变取值范围，从这个范围中选择一个随机整数。然而在许多应用程序中，随机数生成器的取值范围在程序的整个生存期中是固定的。这是事实，`java.util.Random` 类的取值范围正好是整数类型的范围。我们需要这样的一个类，如同 Java 的 `Random` 类一样有一个范围，但不同之处在于这个取值范围是允许客户指定的。下面将要实现的 `RandomIntInRange` 类提供了这一行为，无论何时调用该类的 `nextInt` 方法，它都从固定范围中返回一个随机数，而没有参数传递给 `nextInt`。

当你创立一个新的 `RandomIntInRange` 对象时，提供参数给构造器来指定所需的范围。由于这个范围是类的特性，所以在它的生存期内查询和修改对象的范围是可能的。

简单起见，这里就直接介绍这个类的实现，而没有先讨论它的说明。首先考虑存储结构。`RandomIntInRange` 是组合对象，它唯一的组件是一个 `RandomInt` 对象，保存在 `rnd` 实例域中。另外 `RandomIntInRange` 对象定义了整型的 `min` 和 `max` 域，保存固定范围的最小值和最大值。下面是这个类的类定义：

```
public class RandomIntInRange {

    protected RandomInt rnd;
    protected int min, max;

    public RandomIntInRange(int min, int max)
        throws IllegalArgumentException {
        // EFFECTS: If max < min throws
        //    IllegalArgumentException; else initializes this
    }
}
```

```

// to the range [min..max] and seeds this
// with the system time.
if (max < min) throw new IllegalArgumentException();
rnd = new RandomInt();
setRange(min, max);
}

public RandomIntInRange(Range range)
throws NullPointerException {
// EFFECTS: If range is null throws
// NullPointerException; else initializes this to
// range and seeds with the system time.
this(range.getMin(), range.getMax());
}

public RandomIntInRange(int n)
throw IllegalArgumentException {
// EFFECTS: If n<=0 throws IllegalArgumentException;
// else initializes this to the range [0..n-1]
// and seeds this with the system time.
this(0, n-1);
}

public void setSeed(long seed) {
// MODIFIES: this
// EFFECTS: Reseeds this with seed.
rnd.setSeed(seed);
}

public int nextInt() {
// MODIFIES: this
// EFFECTS: Returns the next random int in the
// current range and updates the state of this
// to reflect the return of this new value.
return rnd.nextInt(min, max);
}

public void setRange(int min, int max)
throws IllegalArgumentException {
// MODIFIES: this
// EFFECTS: If max < min throws
// IllegalArgumentException; else changes the
// current range to [min..max].
this.min = min;
this.max = max;
}

public void setRange(Range range)
throws NullPointerException {
// MODIFIES: this
// EFFECTS: If range is null throws
// NullPointerException; else sets the current
// range to [range.getMin()..range.getMax()].
setRange(range.getMin(), range.getMax());
}

public Range getRange() {
// EFFECTS: Returns the current range.
return new Range(min, max);
}
}

```

练习

4.8 使用以下方法一般化4.2.1节的CoinFlip程序和4.2节的DieRoll程序：写程序NumberToss，它有两个运行参数n和m，程序在固定范围[1..m]中选择产生n个随机数，记录并输出从1到m每个数出现的次数和出现的百分比。你的程序中应该使用RandomIntInRange类。

4.9 使用下面存储结构重新实现RandomIntInRange类：

```
// random-integer generator
protected RandomInt rnd;

// the current range
protected Range rng;
```

确保不改变类的抽象：新的实现应该和前面实现的RandomIntInRange类行为相同。用练习4.8中的NumberToss程序测试你的新的实现版本。

4.2.4 随机点

在这一节设计一个随机点生成器。不言而喻，如同RandomInt类对整数所做的一样，RandomPoint类产生随机点。两者都是可变范围随机数生成器，每次请求一个新的随机值时可指定范围。对RandomPoint类而言，当请求一个新的随机点时，以定义矩形的参数调用NextPoint方法，该调用返回的新的随机点保证在参数决定的矩形内。下面是这个类的类框架：

```
public class RandomPoint {

    public RandomPoint()
        // EFFECTS: Seeds this new object with values
        // based on system time.

    public void setSeedX(long seed)
        // MODIFIES: this
        // EFFECTS: Reseeds the x coordinate generator.

    public void setSeedY(long seed)
        // MODIFIES: this
        // EFFECTS: Reseeds the y coordinate generator.

    public PointGeometry nextPoint()
        // MODIFIES: this
        // EFFECTS: Returns the next random point and
        // updates the state of this to reflect the
        // return of this new value.

    public PointGeometry nextPoint(int m, int n)
        throws IllegalArgumentException
        // MODIFIES: this
        // EFFECTS: If m or n are nonpositive throws
        // IllegalArgumentException; else returns the next
        // random point in the rectangle [0..m-1]x[0..n-1]
        // and updates the state of this.

    public PointGeometry nextPoint(Range xRange,
                                  Range yRange)
        throws NullPointerException
```

```

// MODIFIES: this
// EFFECTS: If xRange or yRange is null throws
//   NullPointerException; else returns the next
//   random point in the rectangle xRange x yRange
//   and updates the state of this.

public PointGeometry nextPoint(RectangleGeometry bnds)
    throws NullPointerException
// MODIFIES: this
// EFFECTS: If bnds is null throws
//   NullPointerException; else returns the next
//   random point in the rectangle bnds and
//   updates the state of this.
}

RandomPoint类是一个组合类，它有两个RandomInt类型组件，其中一个用来生成随机x坐标，另一个生成随机y坐标：
```

```

// fields of RandomPoint class
protected RandomInt xRnd, yRnd;
```

无参数构造器中创建了两个RandomInt对象，分别表示x坐标随机生成器xRnd和y坐标随机生成器yRnd。它们各自的种子应该不同，否则它们将产生同一整数序列，因而产生的点将不是随机的。为使它们的种子不同（如果相同，则生成相同的序列），xRnd使用当前时间作为种子，yRnd用当前时间加上固定的偏移量为种子：

```

// class field of RandomPoint class
final static int SeedOffset = 123; // any positive int

public RandomPoint() {
    xRnd = new RandomInt();
    yRnd = new RandomInt();
    yRnd.setSeed(System.currentTimeMillis() + SeedOffset);
}
```

类中同时提供了分别对x和y坐标生成器播种的方法：

```

// methods of RandomPoint class
public void setSeedX(long seed) {
    xRnd.setSeed(seed);
}

public void setSeedY(long seed) {
    yRnd.setSeed(seed);
}
```

nextPoint方法用于获得下一个随机点，如果不带参数调用nextPoint方法，产生的点的坐标可以在任何位置（点坐标在整个整型数区域）：

```

// method of RandomPoint class
public PointGeometry nextPoint() {
    int x = xRnd.nextInt();
    int y = yRnd.nextInt();
    return new PointGeometry(x, y);
}
```

nextPoint方法还可以以定义矩形的参数列表调用，在每种情况下，由该方法返回的新的随机点保证在参数列表决定的矩形中。矩形可以以三种方式描述，正如下面三个nextPoint版本所展示的：

```

// methods of RandomPoint class
```

```

public PointGeometry nextPoint(int m, int n)
    throws IllegalArgumentException {
    int x = xRnd.nextInt(m);
    int y = yRnd.nextInt(n);
    return new PointGeometry(x, y);
}

public PointGeometry nextPoint(Range xRange, Range yRange)
    throws NullPointerException {
    int x = xRnd.nextInt(xRange);
    int y = yRnd.nextInt(yRange);
    return new PointGeometry(x, y);
}

public PointGeometry nextPoint(RectangleGeometry bounds)
    throws NullPointerException {
    return nextPoint(bounds.xRange(), bounds.yRange());
}

```

练习

4.10 编写一个Java程序（非图形）`SomeRandomPoints`，有一个参数`n`，生成并输出`n`个包含在矩形区域 $[0..100] \times [0..100]$ 中的随机点。

4.11 以3.5节的`MyGraphicsProgram`为模板，编写一个Java图形程序`PaintManyPoints`，它有一个参数`n`。程序实现用`n`个蓝色的随机点填充框架区域。可以用下面这样的域存储结构保存`n`个随机点：

```
protected PointGeometry[] points;
```

为确保随机点包括在框架区域中，先要获得框架内容区域范围，用它构造一个限定范围的矩形区域，然后调用`nextPoint`方法时作为传入。你可以用下面的代码构造限定范围的`bounds`（这里`nbrPoints`是产生的随机点的数量）。`makeContent`的实现中的其他内容由你来写，主要包括生成`n`个随机点并存储在数组`points`中：

```

// method of PaintManyPoints class
public void makeContent() {
    points = new PointGeometry[nbrPoints];
    Dimension d = getFrame().getContentSize();
    RectangleGeometry bounds =
        new RectangleGeometry(0, 0, d.width, d.height);
    ...
}

```

当然，你的`PaintManyPoints`程序也需要实现`MyGraphicsProgram`模板中的其他元素。如果你在这个程序实现上有困难，看过4.2.6节以后再来完成它。

4.12 修改`RandomPoint`类的构造器的实现，使随机数生成器`yRnd`的种子不重置：

```

// constructor: version 2 (faulty)
public RandomPoint() {
    xRnd = new RandomInt();
    yRnd = new RandomInt();
}

```

使用这个版本的`RandomPoint`类，重新编译运行练习4.11的`PaintManyPoints`程序，看一下点还随机分布在框架中吗？如果没有，为什么？（如果你还没有做练习4.11，请用练习4.10中的程序。）

4.13 像实现类RandomIntInRange一样，设计实现RandomPointInRectangle类，在一个固定矩形区域中产生随机点。固定矩形区域由类构造器的参数指定，固定矩形区域是类的特性，客户可在随机点生成器的生存期中访问和修改。下面是这个类的类框架：

```

public class RandomPointInRectangle {

    public RandomPointInRectangle(Range xRange,
                                   Range yRange)
        throws NullPointerException
    // EFFECTS: If xRange or yRange is null throws
    // NullPointerException; else initializes this
    // to the bounding rectangle xRange x yRange
    // and seeds this with values based on
    // the system time.

    public RandomPointInRectangle(RectangleGeometry bnds)
        throws NullPointerException
    // EFFECTS: If bnds is null throws
    // NullPointerException; else initializes this
    // to the bounding rectangle bnds and seeds
    // this with values based on the system time.

    public void setSeedX(long seed)
    // MODIFIES: this
    // EFFECTS: Reseeds the x component generator.

    public void setSeedY(long seed)
    // MODIFIES: this
    // EFFECTS: Reseeds the y component generator.

    public PointGeometry nextPoint()
    // MODIFIES: this
    // EFFECTS: Returns the next random point in the
    // current bounding rectangle and updates the
    // state of this to reflect the return of this
    // new value.

    public void setBounds(RectangleGeometry bnds)
        throws NullPointerException
    // MODIFIES: this
    // EFFECTS: If bnds is null throws
    // NullPointerException; else sets the bounding
    // rectangle to bnds.

    public void setBounds(Range xRange, Range yRange)
        throws NullPointerException
    // MODIFIES: this
    // EFFECTS: If xRange or yRange is null throws
    // NullPointerException; else sets the bounding
    // rectangle to xRange x yRange.

    public RectangleGeometry getBounds()
    // EFFECTS: Returns the current bounding rectangle.
}

```

修改练习4.11的程序，用RandomPointInRectangle类代替RandomPoint类。

4.14 Java中用类java.awt.Color表示颜色，下面是该类的构造器的头：

```
public Color ( int red, int green, int blue )
```

用三个范围在0到255之间整型参数调用Color的构造器，就会创建一个新颜色对象，三个颜色值表示各个颜色成分的亮度（red,green,blue）。例如，下面不同的语句创建不同的颜色：

```
new Color(255, 0, 0)      // red
new Color(0, 0, 255)      // blue
new Color(255, 255, 0)    // magenta
new Color(0, 255, 255)    // cyan
new Color(255, 255, 0)    // yellow
new Color(0, 0, 0)        // black
new Color(127, 127, 127)  // medium gray
new Color(255, 255, 255)  // white
new Color(255, 214, 0)    // gold
new Color(255, 191, 204)  // pink
new Color(64, 105, 207)   // royal blue
```

设计一个RandomColor类来生成随机颜色。你的类应该使用三个随机整数生成器生成在0到255之间的整数，分别对应红、绿、蓝。下面是类框架：

```
public class RandomColor {

    public RandomColor()
        // EFFECTS: Initializes this with seeds
        // based on the system time.

    public void setSeeds(long redSeed, long greenSeed,
        long blueSeed)
        // MODIFIES: this
        // EFFECTS: Reseeds the red, green, and blue
        // random color-component generators.

    public Color nextColor()
        // MODIFIES: this
        // EFFECTS: Returns the next random color and
        // updates the state of this to reflect the
        // return of this new value.
}
```

使用RandomColor类，修改练习4.11中的画随机点程序，使每个点都用一种随机颜色输出。这会有两种情况。第一种情况，不保存每个点的颜色，在paintComponent方法每次被调用时生成随机颜色，然后用它来画点。另一种情况，你希望在产生随机点时生成随机颜色，这样需要将颜色保存在一个数组中（如color[i]存储随机点points[i]的颜色）。

顺便提一下（与本练习无关），Color类用静态域定义了各种基本颜色，静态域在Color类载入时自动创建，然后通过类名加上静态域名直接使用，无需创建对象。Color类是这样定义静态域的：

```
public static final Color green;
```

允许通过用Color.green直接使用绿色。其他由Color类提供的基本颜色有：黑，蓝，青，黑灰，灰，绿，亮灰，紫红，桔红，分红，红，白和黄。

4.2.5 随机矩形

这一节讲述RandomRectangle类，用它来生成随机矩形。如同随机点的位置被限制在一个矩形中一样，随机矩形要限制它的位置和大小。这里用限定矩形来保证生成的随机矩形在某个限定区域中。

除限定矩形之外，随机矩形生成器还应该有宽度范围（width range）特性和高度范围（height range）特性，用它们来保证生成的随机矩形在生成器要求的范围中。矩形的宽度域保证要在生成器的宽度范围的最小和最大值之间，同样高度域也如此。默认的宽度范围从1到限定矩形的宽度，默认的高度范围从1到限定矩形的高度。这就说明默认情况下，随机矩形的尺寸只受限定矩形的限制。生成器的宽度范围特性和高度范围特性是可以查询和修改的。

下面是RandomRectangle类的说明：

```
public class RandomRectangle {

    public RandomRectangle(Range xRange, Range yRange)
        throws NullPointerException
    //EFFECTS: If xRange or yRange are null throws
    //  NullPointerException; else initializes this with
    //  the default width range and height range,
    //  sets the bounding rectangle to xRange x yRange
    //  and seeds this with values based on system time.

    public RandomRectangle(RectangleGeometry bnds)
        throws NullPointerException
    // EFFECTS: If bnds is null throws
    //  NullPointerException; else initializes this with
    //  the default width range and height range,
    //  sets the bounding rectangle to bnds, and seeds
    //  this with values based on the system time.

    public void setSeeds(long sdW, long sdH,
        long sdX, long sdY)
    // MODIFIES: this
    // EFFECTS: Reseeds the generators for width, height,
    //  and x and y coordinates of position.

    public void setWidthRange(Range newRange)
        throws NullPointerException, IllegalArgumentException
    // MODIFIES: this
    // EFFECTS: If newRange is null throws
    //  NullPointerException; else if newRange.getMin()
    //  is nonpositive or newRange.getMax() exceeds the
    //  bounding rectangle's width throws
    //  IllegalArgumentException; else sets the
    //  width range property to newRange.

    public Range getWidthRange()
    // EFFECTS: Returns the current width range.

    public void setHeightRange(Range newRange)
    // MODIFIES: this
    // EFFECTS: If newRange is null throws
    //  NullPointerException; else if newRange.getMin()
    //  is nonpositive or newRange.getMax() exceeds the
    //  bounding rectangle's height throws
```

```

// IllegalArgumentException; else sets the
// height range property to newRange.
public Range getHeightRange()
    // EFFECTS: Returns the current height range.

public RectangleGeometry bounds()
    // EFFECTS: Returns the bounding rectangle.

public RectangleGeometry nextRectangle()
    // MODIFIES: this
    // EFFECTS: Returns the next random rectangle
    // enclosed by the bounding rectangle whose width
    // and height constrained by the current width
    // range and height range, and updates the state of
    // this to reflect the return of this new value.
}

```

这个类框架结构说明与RandomRectangle对象相关的限定矩形在对象被创建时就已经确定，并且在对象的生存期中不能改变。相反，对象的宽度范围和高度范围是可查询和修改的。由于客户在设定高度和宽度范围时可能需要了解限定矩形的大小，来确保它们的高度和宽度范围不会太大，所以类提供了bounds方法用来获得限定矩形。换句话说，调用bounds方法可以保证客户用合理的参数调用setWidthRange和setHeightRange方法。

后面我们将会设计一个用RandomRectangle类，在随机位置，用随机尺寸和颜色在窗口中绘制随机矩形的图形程序。现在先看一段简短的代码熟悉这个类的用法：在 $[0..100] \times [0..200]$ 的限定矩形中，输出10个随机矩形，每一个的宽度在20到40之间，高度在30到60之间。下面是这段代码：

```

RectangleGeometry bounds =
    new RectangleGeometry(new Point(0, 0), 100, 200);
RandomRectangle rndRect = new RandomRectangle(bounds);
rndRect.setWidthRange(new Range(20, 40));
rndRect.setHeightRange(new Range(30, 60));
for (int i = 0; i < 10; i++)
    System.out.println(rndRect.nextRectangle());

```

我们下面逐步实现RandomRectangle类。限制矩形用RandomRectangle对象表示，保存在bounds域中；高度和宽度范围都用RandomIntInRange随机生成器生成，分别保存在rndWidth和rndHeight域中；随机矩形的位置由随机点生成器生成：

```

// fields of RandomRectangle class
// bounding rectangle
protected RectangleGeometry bounds;
// generator for random rectangle positions
protected RandomPoint rndPnt;
// generators for random widths and heights
protected RandomIntInRange rndWidth, rndHeight;

```

这个类使用了几个随机数生成器，它们都由构造器置种子。为保证它们使用不同的种子，下面定义了几个静态常量加到系统时间中产生一个种子，这些种子值以增序给出，否则它们是任意值：

```

// class fields of RandomRectangle class
final static int SeedOffset1 = 4567, SeedOffset2 = 5678,
    SeedOffset3 = 6789;

```

第一个构造器有两个参数，都为Range对象，用来确定限制矩形的大小。在构造器中初始化组成存储结构的域，并且以默认的种子值播种所用的随机对象生成器：

```
public RandomRectangle(Range xRange, Range yRange)
    throws NullPointerException {
    bounds = new RectangleGeometry(xRange, yRange);
    rndWidth =
        new RandomIntInRange(new Range(1, xRange.length()));
    rndHeight =
        new RandomIntInRange(new Range(1, yRange.length()));
    rndPnt = new RandomPoint();
    long time = System.currentTimeMillis();
    rndHeight.setSeed(time + SeedOffset1);
    rndPnt.setSeedX(time + SeedOffset2);
    rndPnt.setSeedY(time + SeedOffset3);
}
```

第二个构造器利用第一个构造器完成：

```
public RandomRectangle(RectangleGeometry bounds)
    throws NullPointerException {
    this(bounds.xRange(), bounds.yRange());
}
```

客户调用setSeeds方法能够重新播种所有随机数生成器的种子：

```
// method of RandomRectangle class
public void setSeeds(long seedW, long seedH,
    long seedX, long seedY) {
    rndWidth.setSeed(seedW);
    rndHeight.setSeed(seedH);
    rndPnt.setSeedX(seedX);
    rndPnt.setSeedY(seedY);
}
```

下面五个方法比较容易实现。第一个方法bounds返回限定矩形，后四个方法是高度范围和宽度范围特性的设置和查询，每个都委托它的工作给相应的组件处理：

```
// methods of RandomRectangle class
public RectangleGeometry bounds() {
    return new RectangleGeometry(this.bounds);
}

public void setWidthRange(Range newRange)
    throws NullPointerException,
        IllegalArgumentException {
    if ((newRange.getMin() <= 0) ||
        (newRange.getMax() > bounds().xRange().length()))
        throw new IllegalArgumentException();
    rndWidth.setRange(newRange);
}

public Range getWidthRange() {
    return rndWidth.getRange();
}

public void setHeightRange(Range newRange)
    throws NullPointerException,
        IllegalArgumentException {
    if ((newRange.getMin() <= 0) ||
        (newRange.getMax() > bounds().yRange().length()))
        throw new IllegalArgumentException();
}
```

```

    rndHeight.setRange(newRange);
}

public Range getHeightRange() {
    return rndHeight.getRange();
}

```

最后转到**nextRectangle**方法，它返回一个新随机矩形。该方法的具体实现是：先生成随机矩形的宽度 w 和高度 h ，然后用这两个值计算插入矩形（*inset rectangle*）的尺寸。插入矩形是在限制矩形内，是宽 w 高 h 的矩形可以被安全定位以确保它不超出限制矩形的边的区域。换句话说，如果一个 $w \times h$ 的矩形的左上角出现在插入矩形里，这个矩形保证在限制矩形内。用插入矩形产生矩形的左上角随机点**pos**。最后该方法在**pos**位置创建并返回一个新的 $w \times h$ 矩形。下面是详细代码：

```

// method of RandomRectangle class
public RectangleGeometry nextRectangle() {
    int w = rndWidth.nextInt();
    int h = rndHeight.nextInt();
    int dw = bounds.getWidth() - w;
    int dh = bounds.getHeight() - h;
    RectangleGeometry insetRectangle =
        new RectangleGeometry(bounds.getPosition(), dw, dh);
    PointGeometry pos = rndPnt.nextPoint(insetRectangle);
    return new RectangleGeometry(pos, w, h);
}

```

练习

4.15 下面是一个简单但不正确的**nextRectangle**方法的实现，使用的存储结构与上面一样：

```

// method of RandomRectangle class: version 2 (faulty)
public RectangleGeometry nextRectangle() {
    PointGeometry pos = rndPnt.nextPoint(bounds);
    int w = rndWidth.nextInt();
    int h = rndHeight.nextInt();
    return new RectangleGeometry(pos, w, h);
}

```

解释这一**nextRectangle**方法的错误版本所采用的方法，并说明这一版本为什么是错误的？

4.2.6 画多个矩形

这一节将在3.5节中**MyGraphicsProgram**模板的基础上设计一个图形程序**PaintManyRectangles**，在窗口中输出 n 个位置、大小、颜色都随机的矩形。下面将逐步讲解这个程序。

先看程序的运行参数要求。随机矩形的尺寸是有限制的，每个矩形的长和宽在一个给定的范围 $[minLen..maxLen]$ （ $minLen$ 不大于 $maxLen$ ）内。调用该程序的命令行格式如下：

```
> java PaintManyRectangles [n [minLen maxLen]]
```

如果不带参数调用，则使用默认值；如果带一个参数 n ，则生成 n 个默认尺寸的随机矩形；如果三个参数都有，后两个参数指定矩形的长和宽的范围。 $n, minLen, maxLen$ 的值在

下面三个类域中保存，并设置初始默认值：

```
// class fields of PaintManyRectangles class
protected static int nbrRectangles = 100;
protected static int minLength = 100, maxLength = 200;
```

如果程序执行时有参数传入，则 `parseArgs` 方法会覆盖上面域的默认值。`parseArgs` 方法的定义为：

```
// class method of PaintManyRectangles class
public static void parseArgs(String[] args) {
    if ((args.length == 2) || (args.length > 3)) {
        String s="USAGE: java PaintManyRectangles [n [min max]]";
        System.out.println(s);
        System.exit(0);
    }
    if (args.length > 0)
        nbrRectangles = Integer.parseInt(args[0]);
    if (args.length > 2) {
        minLength = Integer.parseInt(args[1]);
        maxLength = Integer.parseInt(args[2]);
    }
}
```

下面看一下管理该图形的程序，我们把生成的随机矩形保存在下面的数组中：

```
// field of PaintManyRectangles class
protected RectangleGeometry[] rectangles;
```

`makeContent` 方法创建随机矩形并保存在数组 `rectangles` 中：

```
// method of PaintManyRectangles class
public void makeContent() {
    // step 1: construct a random rectangle generator
    Dimension d = getFrame().getContentSize();
    Range frameWidth = new Range(0, d.width);
    Range frameHeight = new Range(0, d.height);
    RandomRectangle rndRect =
        new RandomRectangle(frameWidth, frameHeight);
    // step 2: set width and height range properties
    Range minMaxLengths = new Range(minLength, maxLength);
    rndRect.setWidthRange(minMaxLengths);
    rndRect.setHeightRange(minMaxLengths);
    // step 3: generate and save random rectangles
    rectangles = new RectangleGeometry[nbrRectangles];
    for (int i = 0; i < nbrRectangles; i++)
        rectangles[i] = rndRect.nextRectangle();
}
```

`makeContent` 方法分三个步骤完成。第一步，创建一个随机矩形生成器 `rndRect`，以框架内容区域的大小初始化以确保每个随机矩形都在框架内。第二步，设置随机矩形生成器的长度范围和宽度范围特性，这些特性限制了生成器生成的矩形的尺寸。第三步，创建长度为 `nbrRectangles` 的 `RectangleGeometry` 类型数组 `rectangles`，然后逐个把新生成随机矩形保存在这个数组。

最后我们实现 `paintComponent` 方法，它用随机颜色绘制存储在数组 `rectangles` 中的矩形：

```
// method of PaintManyRectangles class
public void paintComponent(Graphics g) {
```

```

super.paintComponent(g);
Graphics2D g2 = (Graphics2D)g;
RandomColor rndClr = new RandomColor();
for (int i = 0; i < rectangles.length; i++) {
    // use the next random color for rendering
    g2.setPaint(rndClr.nextColor());
    // paint rectangle r in the current color
    RectangleGeometry r = rectangles[i];
    g2.fill(r.shape());
}
}

```

练习

4.16 完整实现并运行PaintManyRectangles程序。

4.17 目前版本的PaintManyRectangles程序在每次调用paintComponent方法时都会生成新的随机颜色绘制图形。程序框架最小化或最大化时都调用paintComponent方法，那么随机矩形的颜色也随着不断改变（试一下）。修改这个程序的实现，通过makeContent方法使每个矩形有固定的随机颜色。这需要定义颜色数组：

```

protected Color[] colors

```

来保存对应的颜色，colors[i]存储rectangle[i]的颜色。

4.18 RandomColor类（练习4.14中）产生的颜色是不透明的：当你用不透明颜色填充某个图形的时候，无论它下面是否有图形，这个图形都覆盖在上面。用透明或半透明颜色就不会有这种问题。Java的java.awt.Color类提供了构造这种透明颜色的方法：

to parallel the array of rectangles: colors[i] stores the color of

参数alpha取[0..255]范围内的值。如果alpha值为0，完全透明；255则不透明，中间的值主角从透明逐渐到不透明。例如，new Color(255, 0, 255, 220)产生一个稍微透明的紫红色。

定义生成随机半透明颜色的类RandomColorInAlphaRange，这个类定义一个alpha范围特性，随机alpha值在这个范围内生成。红、绿和蓝色组件同RandomColor类中一样随机产生。下面是这个类的类框架：

```

public class RandomColorInAlphaRange {

    public RandomColorInAlphaRange(int minAlpha,
                                    int maxAlpha)
        throws IllegalArgumentException
    // EFFECTS: If 0 <= minAlpha <= maxAlpha <= 255
    // initializes this with the alpha range
    // [minAlpha..maxAlpha] and seeds this based
    // on system time; else
    // throws IllegalArgumentException.

    public RandomColorInAlphaRange(Range aRng)
        throws NullPointerException,
               IllegalArgumentException
    // EFFECTS: If aRng is null throws
    // NullPointerException; else if
    // 0 <= aRng.getMin() and aRng.getMax() <= 255
    // initializes this with aRng and seeds this
    // based on the system time; else
}

```

```

    // throws IllegalArgumentException.

public RandomColorInAlphaRange()
    // EFFECTS: Initializes this with the alpha
    // range [0..255] and seeds this based on
    // the system time.

public void setSeeds(long redSeed, long greenSeed,
                     long blueSeed, long alphaSeed)
    // MODIFIES: this
    // EFFECTS: Reseeds the red, green, blue, and
    // alpha random color-component generators.

public void setAlphaRange(int minAlpha,
                         int maxAlpha)
    throws IllegalArgumentException
    // MODIFIES: this
    // EFFECTS: If 0 <= minAlpha <= maxAlpha <= 255
    // sets alpha range to [minAlpha..maxAlpha];
    // else throws IllegalArgumentException.

public void setAlphaRange(Range newARng)
    throws NullPointerException,
           IllegalArgumentException
    // MODIFIES: this
    // EFFECTS: If newARng is null throws
    // NullPointerException; else if
    // 0 <= newARng.getMin() <= newARng.getMax()
    // <= 255 sets alpha range to newARng;
    // else throws IllegalArgumentException.

public Range getAlphaRange()
    // EFFECTS: Returns current alpha range.

public Color nextColor()
    // MODIFIES: this
    // EFFECTS: Returns new random color whose
    // alpha component lies in the alpha range
    // and updates the state of this to
    // reflect the return of this new value.
}

```

4.19 修改本节的PaintManyRectangles程序，使它产生的随机矩形由随机透明颜色填充。运行时命令行采用如下格式：

```
> java PaintManyRectangles [n [minLen maxLen [minAlpha maxAlpha]]]
```

前面三个参数含义同原来程序，第四和第五个参数为随机颜色的alpha范围，如果这些参数不给出，默认alpha范围为[0..255]。

4.3 多组件组合

到目前为止我们所看到的组合体类都有一个共性：组件的数目少且事先确定。然而有时包含在一个组合体的组件数目虽固定但有可能相当大。在这种情况下，不可能一个一个地保存组件，明智的做法是定义一个组件集合（collection）对象，由它来保存这些组件，而不是为每个组件声明一个独立的域。游戏Monopoly包括28个特性（模拟真实状态），但

我们不会定义一个Monopoly类，声明28个名为atlanticAvenue、shortlineRailroad、parkplace之类的域。

还有一种情况是，组合体中组件的数目在运行前并不清楚，或者在运行中可以改变。这样就不能定义确定数目的域保存它们。一个菜单有可变数目的项目、一幅图由可变数目的图片组成、一个消防队包括数目可变的消防车，类似这样的组合体对象应该用集合对象保存组件。

除了方便之外，使用集合对象还有很多优点。集合对象可以给组合体提供操作管理组件的服务：包括不同的访问操作、插入新元素、删除元素和其他操作。利用这些服务，组合体对象的实现变得更加简单。此外，组合体对象也可以提供相似的接口服务，并通过把这些对服务的请求委托给它的组件来实现。

让我们看一下Vector类，它是处理集合的Java标准类之一。我们将把这个类放在表示折线的类的实现中。

4.3.1 Java的Vector类

java.util.Vector类就是这样一个集合类，它实现随着容纳新项的需要而增长的对象列表。像数组一样，矢量（vector）按顺序维持一个对象列表，并允许按下标存取对象；与数组不同的是，矢量允许在任何位置插入新项，原来位置的项和它后面的项依次后移一个位置。如果一个新项被插入到位置5，原来在位置5的项移到位置6，原来在位置6的项移到位置7，这样一直向后移动。同样，当一个项从矢量中删除时，它后面的每个项都要向前移动一个位置。注意这个行为不是类实现细节，但它和项如何索引有关，也和客户如何存取有关，即和矢量的接口有关。

下面的类框架仅仅列出我们要用到的操作，你可以查阅Java文档了解更完整的内容：

```
public class java.util.Vector {

    public Vector()
        // EFFECTS: Initializes this to a new empty vector.

    public int size()
        // EFFECTS: Returns the number of items stored in
        // this vector.

    public Object get(int i)
        throws ArrayIndexOutOfBoundsException
        // EFFECTS: If 0 <= i < size() returns the item
        // stored at index i; else throws
        // ArrayIndexOutOfBoundsException.

    public void set(int i, Object newObj)
        throws ArrayIndexOutOfBoundsException
        // MODIFIES: this
        // EFFECTS: If 0 <= i < size() replaces the item
        // at index i with newObj; else throws
        // ArrayIndexOutOfBoundsException.

    public void add(Object newObj)
        // MODIFIES: this
        // EFFECTS: Adds newObj to the end of this vector
        // (at the index size()), increasing the size of
        // this vector by one.
}
```

```

public void add(int i, Object newObj)
    throws ArrayIndexOutOfBoundsException
// MODIFIES: this
// EFFECTS: If 0 <= i <= size(), inserts newObj at
// index i and increases by one the index of every
// item whose index had been equal to or greater
// than i, and increases the size of this vector by
// one; else throws ArrayIndexOutOfBoundsException.

public boolean remove(Object obj)
// MODIFIES: this
// EFFECTS: If some element in this is equal to obj,
// removes such an element of least index i and
// returns true, while decreasing by one the index
// of every item whose index had been greater than
// i and decreasing the size of this vector by one;
// else returns false.

public Object remove(int i)
    throws ArrayIndexOutOfBoundsException
// MODIFIES: this
// EFFECTS: If 0 <= i < size(), removes and returns
// the item at index i and decreasing by one the
// index of every item whose index had been greater
// than i and decreases the size of this vector by
// one; else throws ArrayIndexOutOfBoundsException.

public void clear()
// MODIFIES: this
// EFFECTS: Removes all of the elements from this
// vector.

public int indexOf(Object obj)
// EFFECTS: Returns the index of the first occurrence
// of some item in this vector that is equal to
// obj; else if obj does not occur in this vector
// returns -1.
}

```

矢量的管理项定义为 `java.lang.Object` 类型 (`Object` 类是所有类的父型), 因此任何类型的对象都可以保存在矢量中。实际上, 矢量常被用来存储相同类型的对象或比 `Object` 更常用的其他父型的对象。所以在获取矢量元素时, 习惯用强制转换把返回对象转换成它实际的类型或有用的父型:

```
SomeType myObject = (SomeType)vector.get(someGoodIndex);
```

矢量不能存储原始类型对象, 如整数或浮点数; 然而 Java 为每一种原始类型提供了一个包装类 (wrapper class), 用它把数值变成等价的对象。例如, 下面的代码段包装了整数 7, 并把它保存在矢量 `v` 的 `indx` 位置中:

```
Integer iObj = new Integer(7);
v.add(indx, iObj);
```

用下面一条语句可以从矢量 `v` 的 `indx` 位置中取出 `Integer` 对象, 并从整型对象中取得该对象的值:

```
int i = ((Integer)v.get(indx)).intValue(); // now i==7
```

组合体对象使用矢量来保存组件, 下面将设计 `PolylineGeometry` 类来看它的用法。

在这个例子中，矢量用来保存折线的顶点。

4.3.2 折线

顾名思义，折线（polyline）是由平面上的若干条直线段组成的，这些直线段由端点连在一起。线段称为折线的边（edge），线段的端点叫做折线的顶点（vertex）。折线的第一个和最后一个顶点只有一条边相连，而其他所有的顶点都和两条边相连。一条具有 n 个顶点的折线有 $n-1$ 条边。在特殊情况下，一条折线可以只包含一个点而没有边。折线的边可以相交。另外，如果两个顶点正好重合，它们仍被视为两个不同的顶点。

图4-4示意了对折线元素索引编号的方法。 n 个顶点依次为从0到 $n-1$ ，标识为从 v_0 到 v_{n-1} ； $n-1$ 条边被依次编为从0到 $n-2$ ，使得边 e_i 所连接的点恰好是 v_i 和 v_{i+1} 。后面的表示折线的类就采用这种编号方法对折线的顶点和边进行访问。

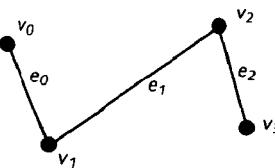


图4-4 一条折线

下面是折线类PolylineGeometry的框架：

```
public class PolylineGeometry {

    public PolylineGeometry(PointGeometry[] vs)
        throws NullPointerException, ZeroArraySizeException
    // EFFECTS: If vs is null or vs[i] is null for some
    // legal i throws NullPointerException; else if
    // vs.length==0 throws ZeroArraySizeException;
    // else initializes this to the vertex sequence
    // given by vs.

    public PolylineGeometry(PolylineGeometry poly)
        throws NullPointerException
    // EFFECTS: If poly is null throws
    // NullPointerException; else initializes this
    // to be a copy of poly.

    public PointGeometry getVertex(int i)
        throws IndexOutOfBoundsException
    // EFFECTS: If 0 <= i < nbrVertices() returns the
    // position of the vertex at index i;
    // else throws IndexOutOfBoundsException.

    public void setVertex(int i, PointGeometry v)
        throws NullPointerException, IndexOutOfBoundsException
    // MODIFIES: this
    // EFFECTS: If v is null throws NullPointerException;
    // else if 0<=i<nbrVertices() changes the position
    // of the vertex at index i to v;
    // else throws IndexOutOfBoundsException.

    public LineSegmentGeometry edge(int i)
        throws IndexOutOfBoundsException
}
```

```

// EFFECTS: If 0<=i<nbrEdges() returns the edge at
// index i; else throws IndexOutOfBoundsException.

public int nbrVertices()
// EFFECTS: Returns the number of vertices in this.

public int nbrEdges()
// EFFECTS: Returns the number of edges in this.

public java.awt.Shape shape()
// EFFECTS: Returns the shape of this polyline.

public void translate(int dx, int dy)
// MODIFIES: this
// EFFECTS: Translates this polyline by dx along x
// and by dy along y.

public String toString()
// EFFECTS: Returns "Polyline: v0,v1,...,vn-1"
// where each vi describes vertex i.
}

```

类中edge方法返回的边由LineSegmentGeometry对象（见练习3.3）表示。当一个PolylineGeometry对象被创建时，它所包含的顶点的个数就确定了。该类提供了改变顶点位置的方法（setVertex方法），但没有提供插入新点以及删除已存在点的方法。下面是一个简单的非图形程序，它演示了PolylineGeometry类的行为：

```

public class TryPolylineGeometry {
    public static void main(String[] args) {
        PointGeometry[] vs = new PointGeometry[] {
            new PointGeometry(0, 0),
            new PointGeometry(10, 0),
            new PointGeometry(10, 10)
        };
        PolylineGeometry poly1 = new PolylineGeometry(vs);
        System.out.println(poly1);
        // Polyline: (0,0),(10,0),(10,10)
        PolylineGeometry poly2 = new PolylineGeometry(poly1);
        poly2.setVertex(0, new PointGeometry(40, 50));
        System.out.println(poly2);
        // Polyline: (40,50),(10,0),(10,10)
        poly2.translate(100, 200);
        System.out.println(poly2);
        // Polyline: (140,250),(110,200),(110,210)
    }
}

```

接下来看PolylineGeometry类的实现。

PolylineGeometry类中折线的顶点的用Vector来保存。折线的顶点按下标排序：出现在矢量的位置i上的点为点 v_i ，它声明如下：

```
// field of PolylineGeometry class
protected Vector vertices;
```

有两种构造折线的方法：可以用包含折线所有顶点的非空数组构造；也可以由一个已存在的折线构造。后一种情况下，新的折线实际上是已存在折线的一个拷贝。

```
public PolylineGeometry(PointGeometry[] vs)
throws NullPointerException, ZeroArraySizeException {
    if (vs.length == 0) throw new ZeroArraySizeException();
```

```

vertices = new Vector();
for (int i = 0; i < vs.length; i++)
    vertices.add(new PointGeometry(vs[i]));
}

public PolylineGeometry(PolylineGeometry poly)
    throws NullPointerException {
    vertices = new Vector();
    for (int i = 0; i < poly.nbrVertices(); i++)
        vertices.add(new PointGeometry(poly.getVertex(i)));
}

```

getVertex方法返回序号为i的顶点的位置：

```

// methods of PolylineGeometry class
public PointGeometry getVertex(int i)
    throws IndexOutOfBoundsException {
    return new PointGeometry((PointGeometry)vertices.get(i));
}

```

当调用getVertex方法时，如果其参数i是一个越界的序号，那么vertices.get方法会抛出ArrayIndexOutOfBoundsException异常。getVertex方法的throws子句所声明的类型IndexOutOfBoundsException是ArrayIndexOutOfBoundsException类型的一个父型，因此throws子句是合法的。

setVertex方法用于在平面上移动折线顶点的位置：

```

// method of PolylineGeometry class
public void setVertex(int i, PointGeometry v)
    throws NullPointerException, IndexOutOfBoundsException {
    vertices.set(i, new PointGeometry(v));
}

```

edge方法返回边 e_i ，它连接着顶点 v_i 和 v_{i+1} 。由于边并不保存在存储结构里，每次调用edge方法时要先构造出边 e_i 再返回它：

```

// method of PolylineGeometry class
public LineSegmentGeometry edge(int i)
    throws IndexOutOfBoundsException {
    return new LineSegmentGeometry(getVertex(i),
        getVertex(i+1));
}

```

获取折线顶点和边的数量的方法是很简单的：

```

// methods of PolylineGeometry class
public int nbrVertices() {
    return vertices.size();
}

public int nbrEdges() {
    return nbrVertices() - 1;
}

```

通过改变每个顶点的位置来移动一条折线，折线类的translate方法使折线沿x轴移动参数dx个单位，沿y轴移动参数dy个单位：

```

// method of PolylineGeometry class
public void translate(int dx, int dy) {
    for (int i = 0; i < nbrVertices(); i++) {

```

```

    PointGeometry p = (PointGeometry)vertices.get(i);
    p.translate(dx, dy);
}
}

```

一条折线的形状是用Java的GeneralPath类创建的，它是java.awt.geom包的一个成员。这个类所记录的形状路径既包括直线段又包括曲线段。为了描述一个通用的路径，我们构造了一个空的GeneralPath对象，然后用它的moveTo方法增加一个初始点。获得下一个点，调用lineTo方法把初始点和这个点用直线段连起来。然后获得下一个点，把上个点和它连起来。这样，通过反复调用lineTo方法一段一段地产生一条路径。下面是shape方法的实现：

```

// method of the PolylineGeometry class
public Shape shape() {
    GeneralPath path = new GeneralPath();
    PointGeometry v = getVertex(0);
    path.moveTo(v.getX(), v.getY());
    for (int i = 1; i < nbrVertices(); i++) {
        v = getVertex(i);
        path.lineTo(v.getX(), v.getY());
    }
    return path;
}

```

toString方法先输出描述类型的字符串“Polyline”，接着按顺序输出折线的顶点。折线顶点的输出是调用保护型的verticesToString方法完成的。它们实现如下：

```

// methods of PolylineGeometry class
public String toString() {
    return "Polyline: " + verticesToString();
}

protected String verticesToString() {
    String res = "";
    for (int i = 0; i < nbrVertices() - 1; i++)
        res += getVertex(i) + ",";
    res += getVertex(nbrVertices() - 1);
    return res;
}

```

练习

4.20 用MyGraphicsProgram模板编写一个图形程序：画一条黄色的折线。程序参数应包含偶数个整数，并且不少于四个：

```
> java PaintPolyline x0 y0 x1 y1 ...
```

折线点 v 的 x 和 y 的坐标分别是由第 $2i$ 个和第 $2i+1$ 个参数决定。

4.21 字典是一个由名字-值对组成的动态集合。其中名字是唯一的，名字是字符串类型；值则可以是任何类型的非空对象。可以通过名字来查找相对应的值，可以插入新的名字-值对，而已存在的名字-值对也可以被删除，也可以用序号来访问名字-值对，序号的范围是从0到字典所含名字-值对的个数减1。名字-值对可以按任何顺序保存，若按序号访问，将得到以任意顺序排列的元素。下面是这个类的类框架：

```
public class Dictionary {
```

```

public Dictionary()
    // EFFECTS: Initializes this to an empty
    // dictionary.

public Object insert(String name, Object value)
    throws NullPointerException
    // MODIFIES: this
    // EFFECTS: If name or value is null throws
    // NullPointerException; else if some pair
    // named by name exists replaces its value
    // by value; else inserts the new pair
    // name-value into this dictionary.

public Object remove(String name)
    throws NullPointerException
    // MODIFIES: this
    // EFFECTS: If name is null throws
    // NullPointerException; else if some pair
    // named by name exists removes it from this
    // dictionary and returns its value;
    // else returns null.

public Object find(String name)
    throws NullPointerException
    // EFFECTS: If name is null throws
    // NullPointerException; else if some pair
    // named by name exists returns its value;
    // else returns null.

public int size()
    // EFFECTS: Returns the number of pairs
    // stored in this dictionary.

public boolean isFull()
    // EFFECTS: Returns true if no more pairs can
    // be inserted into this dictionary;
    // else returns false.

public String name(int i)
    throws IndexOutOfBoundsException
    // EFFECTS: If 0 <= i < size() returns the name
    // of the pair at index i; else throws
    // IndexOutOfBoundsException.

public Object value(int i)
    throws IndexOutOfBoundsException
    // EFFECTS: If 0 <= i < size() returns the value
    // of the pair at index i; else throws
    // IndexOutOfBoundsException.

public Object value(String name)
    throws NullPointerException
    // EFFECTS: If name is null throws
    // NullPointerException; else if some pair
    // named by name exists returns its value;
    // else returns null.
}

```

实现Dictionary类。

[提示：Dictionary是一个组合类，它是由若干个Attribute对象组成的集合（Attribute类在练习3.4中已有描述）。集合中的每个Attribute对象表示字典中的一个名字-值对。注意，所使用的操作一定要同时被Attribute类和你的集合（例如，可以选择Vector对象表示）所支持。但是Dictionary类的客户不需要了解Attribute的任何信息。]

你也许会想到定义一个保护型方法，用以根据给定的名字来查找相应的名字-值对在集合中的序号。如果你将名字-值对保存在一个矢量中，方法可以这样说明：

```
protected int findAttribute(String name)
    // EFFECTS: If a pair named by name exists
    // returns its index; else returns -1.
```

findAttribute方法在实现类的公有方法时是很有用的，举个例子来说，实现remove方法时，可先用findAttribute方法来取得要被删除的元素的序号，再调用vector的remove方法删除这个序号所对应的元素。]

4.22 本题所描述的TryDictionary是一个基于文本的应用程序，它可以通过用户的输入来测试Dictionary类。该应用程序允许用户交互处理名字-值对（其中值是字符串类型的）。下面列出了用户可以在终端输入的命令：

- **insert name value** 将一个name-value插入到字典中；如果以name为名字的名字-值对已经存在，则修改它的值为value。
- **remove name** 将以name为名字的名字-值对删除；如果没有这样的对存在，就什么也不做。
- **find name** 打印出以name为名字的名字-值对的值；如果没有这样的对存在，打印出一条消息。
- **size** 打印出字典的大小。
- **print** 按任意顺序打印出字典中的对。
- **quit** 退出应用程序。

下面是一个简单的交互过程，加粗的字体是用户的输入：

```
> java TryDictionary
? insert banana yellow
? insert apple red
? find banana
yellow
? find artichoke
*** artichoke not found ***
? insert banana blue
? size
2
? print
banana:blue
apple:red
? remove apple
? find apple
*** apple not found ***
? print
banana:blue
? quit
```

4.4 表达一致性约束

具体类为它的类型提供了实现：它为它的对象提供表达规则并且根据选择的表达方式实现类中方法。表达规则描述了如何正确表示这个类型中的值。`PointGeometry`类的实例表示平面上的点：实例的x和y域值分别表示了点的x, y坐标。`RectangleGeometry`类的实例表示平面上的矩形：`RectangleGeometry`的各个域的值分别表示了矩形的位置、宽度和高度。

根据表达规则，可以想象无效的对象是不能表示任何东西的。例如，对于一个`RectangleGeometry`对象，如果表示其宽度的域是一个负数，那这个对象就是无效的，因为矩形的宽度必须是一个非负数。还有，如果一个`PolylineGeometry`对象的`vertices`矢量中根本没有点，那么这个`PolylineGeometry`对象也是无效的，这样的对象是和没有顶点的折线对应的，但是事实上没有这种折线。

本节重点讨论表达一致性约束，它描述了使一个表达有效是什么意思。在概述之后，我们将讨论两种具有表达一致性约束的类：一个是平面中的椭圆类，另一个是有理数类。

4.4.1 概述

一个对象的历史影响它的行为。行为依赖过去事件的原因在于对象有一种记忆方式——状态。不仅是Java对象，很多实际的事物也都是这样的。比如你自己，作为一个人，你会受到你的经历的影响，每一种经历都会在你身上留下它的标记：在你的记忆里、你的身体上、你的性情上以及你的思想上。你所有的经历使你达到了现在的状态，这种状态影响——有时候可以说决定——了你的行为。举一个更普通的例子，对于一台收音机，当你打开它的时候，它所播放的是你上一次调出的电台的节目。电台的调动是收音机状态的一部分。如果你将收音机调到了另一个台，因为状态改变了（它保存了一个新的电台的频率），收音机的行为就会改变（它会播放一个新的电台的节目）。

在Java语言中，对象的状态对应于保存在它的域中的值。当对象响应它所收到的消息时，这些值可能会发生变化，这种变化叫做状态转变（state transition）。我们通过研究在一系列消息的过程中Java对象域发生的变化，来观察对象的历史对其行为的影响。在下面的代码段中，假设`RectangleGeometry`类的实现和在3.2.2节中给出的一样，即一个矩形包含两个Range域：`xRange`和`yRange`：

```
RectangleGeometry r = new RectangleGeometry(2, 2, 4, 5);
    // xRange: [2..6], yRange: [2..7]
r.setPosition(new PointGeometry(8, 9));
    // xRange: [8..12], yRange: [9..14]
r.setWidth(7);    // xRange: [8..15], yRange: [9..14]
r.getHeight();    // xRange: [8..15], yRange: [9..14]
```

有些类型的消息会导致状态的变化（例如`setWidth`方法），有的消息则不会改变状态（`getWidth`方法）。正如在3.2节中提到的，能够导致状态转变的那些方法叫做增变器（mutator），而那些访问状态但不改变它们的方法叫做选择器（selector）。

一个对象的状态只包含那些对于此对象的抽象非常重要的部分。保存一个对象完整的历史通常是没有必要的。正如收音机保存了现在收听的电台，但并没有保存此收音机曾经调到过的所有台。再举一个例子，上一程序段中的`RectangleGeometry`对象`r`的最后状态和下面程序段中的对象`s`状态完全相同：

```
RectangleGeometry s = new RectangleGeometry(8, 9, 7, 5);
// xRange: [8..15], yRange: [9..14]
```

虽然r和s有着不同的历史，但你仅从它们的状态上是看不出这一点。

每个对象的状态都与其他对象不同，包括同一类型的对象。在刚才给出的代码段中，`RectangleGeometry`对象r和s碰巧处在相同的状态，但是如果给它们各发一个状态变化消息，它们的状态就会相互独立地改变。语句：

```
s.setPosition(new PointGeometry(12, 14));
```

改变矩形s的状态但对矩形r没有影响。

对于很多类型的对象，可以认为状态的设置是受到一致性约束（consistency constraint）保护的。一个FM收音机只能在标准的FM频率区域内被调动。同样，`RectangleGeometry`对象的`width`和`height`都必须是非负数。对象的每个方法的定义都保证遵守对象的一致性约束。收音机的调台旋钮将频率限制在收音机可以接受到的范围内。`RectangleGeometry`对象的客户不能用`setWidth`方法将对象的宽度设置成一个负数。语句：

```
s.setWidth(-18);
```

不能将矩形的宽度设置成 -18，它将抛出一个非法参数异常。

同样，信息隐藏是不允许客户通过直接访问对象域来改变其状态。对象通过强迫客户使用它们的公有接口来达到一致性约束。比如，收音机外面的硬塑料外壳阻止用户直接操作收音机内部的电子器件；与此类似，`PolylineGeometry`对象p的客户也不能直接把它的`vertices`置成一个空矢量：

```
p.vertices = new Vector();
```

因为`vertices`域是保护型的。编译器会提示这条赋值语句出现访问错误，至少那些没有权力的客户是不允许访问`PolylineGeometry`的保护型数据的。

一致性约束被表述为表达一致性约束。前置条件和后置条件描述了一个过程的行为，而表达一致性约束描述的是类实例的表达特性。表达一致性约束描述了类实例处于一致的状态是什么意思。`PolylineGeometry`类包含如下的表达一致性约束：

实例域`vertices`是一个包含且只能包含点的向量

相反，`PointGeometry`类的表达一致性约束是很容易满足的，因为域`x`和`y`取任何值都表示平面上的一个点。

当一个对象被创建时，它的表达一致性约束也应被创建，这是类的构造器的职责。另外，任何公有方法退出时，该方法必须保证表达一致性约束被满足。这样类的表达一致性约束给公有方法增加了限制。从另一个方面来看，公有方法也受益于这种约束，因为它们可以理所当然地认为，调用它们时表达一致性约束是被满足的。

当一个类的方法正在被执行时，表达一致性约束不需要被满足。方法向着特定的目标执行：目标之一就是建立它所保证的后置条件；另一个目标是建立表达一致性约束。在执行它的工作的过程中，方法可以违反一个或多个表达一致性约束，就像它也可以违反自己的后置条件一样。但是当方法完成它的工作时，它必须恢复满足表达一致性约束并建立后置条件。

表达一致性约束的作用在于强调信息隐藏的重要性。如果信息隐藏被恰当地使用，可以保证只有那些要求直接访问对象域的客户才有这样的访问权，而其他所有客户都只能通过公有方法来访问。如果方法被正确地实现了，那么对象就会遵守表达一致性约束。

相反，如果信息隐藏被破坏了，客户可以随意直接访问对象域，那么他们可能会改变域值，从而破坏对象的状态。一旦客户能不通过对象的公有接口访问对象域，对象就不能再保证它的表达的有效性了。

4.4.2 椭圆

这一节我们将设计椭圆类，通过类的实现来理解上节讨论的表达一致性约束的思想。

椭圆是平面上点的集合。在这个集合中，任意点 p 到两个固定点的距离之和为定长 D 。这两个固定的点称为椭圆的焦点，它们的集合就是椭圆的焦点集。椭圆的主轴是通过椭圆的两焦点并连接相对两端点的线段。不难看出椭圆主轴的长度等于定长 D ，参见图4-5。

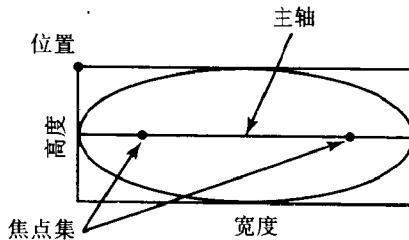


图4-5 椭圆和它的外接矩形

假想平面是一块木板的表面，两个焦点处分别钉上两根钉子，一根长度为 D 的线的两端分别绑缚在两根钉子上。用一支铅笔拉紧绳子并且使笔尖贴在木板上，保持绳子始终拉紧使铅笔围绕焦点转，绘出笔尖的运动轨迹。铅笔绘出的轨迹就是一个椭圆。

我们将讨论的椭圆是在标准方向上的，也就是说，每个椭圆的主轴或者是在垂直方向上，或者是在水平方向上。椭圆的外接矩形是指能够围绕椭圆并且边与轴平行的最小矩形。可以根据椭圆的外接矩形来定义椭圆的位置和大小。椭圆的位置是其外接矩形左上角的顶点，宽度和高度分别是外接矩形的宽和高（如图4-5所示）。和矩形类似，可以把椭圆的位置、宽度、高度作为它的维数。

椭圆类和第3章中的矩形类很相似，主要的不同之处在于椭圆类只提供获得其外接矩形的方法，而没有提供`xRange`或`yRange`方法（椭圆的跨距可以由其外接矩形获得）。下面是`EllipseGeometry`类的说明：

```
public class EllipseGeometry {

    public EllipseGeometry(int x, int y,
                          int width, int height)
        throws IllegalArgumentException
    // EFFECTS: If width or height is negative throws
    // IllegalArgumentException; else initializes this
    // ellipse to position (x,y) and specified
    // width and height.

    public EllipseGeometry(PointGeometry pos,
                          int width, int height)
        throws NullPointerException, IllegalArgumentException
    // EFFECTS: If pos is null throws
    // NullPointerException; else if width or height
    // is negative throws IllegalArgumentException;
    // else initializes this to position pos and
    // specified width and height.
}
```

```

public EllipseGeometry(Range xRange, Range yRange)
    throws NullPointerException
// EFFECTS: If xRange or yRange is null throws
// NullPointerException; else initializes this
// to the dimensions of xRange x yRange.

public EllipseGeometry(EllipseGeometry r)
    throws NullPointerException
// EFFECTS: If r is null throws NullPointerException;
// else initializes this to an ellipse with the
// same dimensions as r.

public PointGeometry getPosition()
// EFFECTS: Returns this ellipse's position.

public void setPosition(PointGeometry p)
    throws NullPointerException
// MODIFIES: this
// EFFECTS: If p is null throws NullPointerException;
// else changes this ellipse's position to
// (p.getX(),p.getY()).

public int getWidth()
// EFFECTS: Returns this ellipse's width.

public void setWidth(int newWidth)
    throws IllegalArgumentException
// MODIFIES: this
// EFFECTS: If newWidth is negative throws
// IllegalArgumentException; else changes this
// ellipse's width to newWidth.

public int getHeight()
// EFFECTS: Returns this ellipse's height.

public void setHeight(int newHeight)
    throws IllegalArgumentException
// MODIFIES: this
// EFFECTS: If newHeight is negative throws
// IllegalArgumentException; else changes this
// ellipse's height to newHeight.

public RectangleGeometry boundingBox()
// EFFECTS: Returns this ellipse's bounding box.

public boolean contains(int x, int y)
// EFFECTS: Returns true if the point (x,y) is
// contained in this ellipse; else returns false.

public boolean contains(PointGeometry p)
    throws NullPointerException
// EFFECTS: If p is null throws NullPointerException;
// else returns true if p is contained in this
// ellipse; else returns false.

public java.awt.Shape shape()
// EFFECTS: Returns the shape of this ellipse.

public void translate(int dx, int dy)

```

```

// MODIFIES: this
// EFFECTS: Translates this ellipse by dx along x
// and by dy along y.

public String toString()
    // EFFECTS: Returns "Ellipse: (x,y),width,height".
}

```

下面来讨论该类的实现。椭圆的表达要比矩形复杂得多，分别用position, width, height来保存椭圆的位置、宽度和高度：

```

// fields of EllipseGeometry class
protected PointGeometry position;
protected int width, height;

```

椭圆两个焦点的位置保存在PointGeometry类型域中，其主轴的长度存在一个整型域中：

```

// fields of EllipseGeometry class
// positions of the two foci
protected PointGeometry focus1, focus2;
// length of the major axis
protected int majorAxis;

```

contains方法有两个参数，其实现要用到上面的三个域。它判断点 $p(x, y)$ 是否在椭圆内，判断方法依据椭圆的定义：当且仅当点 p 到两焦点的距离之和不大于椭圆主轴长度时，该方法返回true。

```

// method of EllipseGeometry class
public boolean contains(int x, int y) {
    PointGeometry p = new PointGeometry(x, y);
    return
        (focus1.distance(p)+focus2.distance(p))<=majorAxis;
}

```

contains方法的正确性依赖于focus1, focus2和majorAxis域值的正确性，而这些值又依赖于椭圆当前的维数，椭圆的维数是在它被创建时初始化的。一旦椭圆的任何维数变化了，这三个域也将相应的被更新。这就是椭圆的表达一致性约束：

保存在域focus1, focus2和majorAxis中的值代表椭圆的焦点和主轴的长度，而保存在域position, width和height中的值代表椭圆的维数。

创建椭圆时，它的表达一致性约束也应被建立，并且在椭圆的生存期内不能被违反。先来看椭圆的构造器：有四个参数的构造器先初始化椭圆的position、width和height域，然后调用保护型方法computeFoci来初始化其他三个域：

```

public EllipseGeometry(int x, int y, int width, int height)
    throws IllegalArgumentException {
    if ((width < 0) || (height < 0))
        throw new IllegalArgumentException();
    this.position = new PointGeometry(x, y);
    this.width = width;
    this.height = height;
    computeFoci();
}

```

computeFoci方法根据椭圆的当前维数设置存储两个焦点位置及主轴长度的域。先计算椭圆的中心点，之后再利用三角学方法计算出两个焦点的位置，椭圆主轴长度等于其宽度和高度中的较大者。

```

// method of EllipseGeometry class
protected void computeFoci() {
    // REQUIRES: Position is not null, and height
    // and width are not negative.
    // EFFECTS: Sets the fields focus1 and focus2
    // to this ellipse's foci, and majorAxis to the
    // length of its major axis.
    double a = getWidth() / 2.0;
    double b = getHeight() / 2.0;
    PointGeometry pos = getPosition();
    int x = (int)(pos.getX() + a);
    int y = (int)(pos.getY() + b);
    int c;
    if (a > b) { // ellipse has horizontal major axis
        c = (int)Math.round(Math.sqrt(a*a - b*b));
        focus1 = new PointGeometry(x + c, y);
        focus2 = new PointGeometry(x - c, y);
        majorAxis = getWidth();
    } else { // ellipse has vertical major axis
        c = (int)Math.round(Math.sqrt(b*b - a*a));
        focus1 = new PointGeometry(x, y + c);
        focus2 = new PointGeometry(x, y - c);
        majorAxis = getHeight();
    }
}

```

`computeFoci`帮助调用它的方法来建立表达一致性约束。由于这个方法不是 `EllipseGeometry`类的公有接口，因此在前面的类框架中没有声明此方法。

其他构造器不需要直接调用`computeFoci`，它们调用上面定义的四个参数的构造器：

```

public EllipseGeometry(PointGeometry pos,
                       int width, int height)
throws NullPointerException, IllegalArgumentException {
    this(pos.getX(), pos.getY(), width, height);
}

public EllipseGeometry(Range xRange, Range yRange)
throws NullPointerException {
    this(xRange.getMin(), yRange.getMin(),
        xRange.length(), yRange.length());
}

public EllipseGeometry(EllipseGeometry e)
throws NullPointerException {
    this(e.getPosition(), e.getWidth(), e.getHeight());
}

```

当创建一个新椭圆时，表达一致性约束就要建立并且在椭圆的整个生存期（每两次调用公有方法之间）中必须遵守。只有三个操作可能会破坏这个约束，它们都用于改变椭圆维数：`setPosition`，`setWidth`，`setHeight`。为了能够在这些方法返回前恢复表达一致性约束，实现这些操作通常的做法是：先调用特定的方法来更新椭圆的位置、宽度或高度，然后再调用`computeFoci`来恢复椭圆的表达一致性。

```

// methods of EllipseGeometry class
public void setPosition(PointGeometry p)
throws NullPointerException {
    position.setX(p.getX());
    position.setY(p.getY());
    computeFoci();
}

```

```

    }

    public void setWidth(int newWidth)
        throws IllegalArgumentException {
        if (newWidth < 0) throw new IllegalArgumentException();
        width = newWidth;
        computeFoci();
    }

    public void setHeight(int newHeight)
        throws IllegalArgumentException {
        if (newHeight < 0) throw new IllegalArgumentException();
        height = newHeight;
        computeFoci();
    }
}

shape方法返回椭圆的形状，该方法定义如下：

// method of EllipseGeometry class
public java.awt.Shape shape() {
    PointGeometry pos = getPosition();
    return new Ellipse2D.Float(pos.getX(), pos.getY(),
        getWidth(), getHeight());
}

```

练习

4.23 实现EllipseGeometry类的其他方法。下面的代码段演示了toString和translate方法的行为：

```

EllipseGeometry e = new EllipseGeometry(2, 3, 7, 8);
System.out.println(e); // Ellipse: (2,3),7,8
e.translate(10, 20);
System.out.println(e); // Ellipse: (12,23),7,8

```

4.24 下面介绍实现EllipseGeometry类的另一种方法：去掉focus1, focus2和majorAxis域以及和它们相关的表达一致性约束，并把两参数的contains方法的实现做以下改变：每当contains方法被调用时，它首先计算出椭圆的焦点和主轴长度，然后用这些值来确定给定点是否在椭圆内部。这个方法的定义采用如下的形式：

```

// method of EllipseGeometry class: version 2
public boolean contains(int x, int y) {
    PointGeometry focus1, focus2;
    int majorAxis;
    // initialize the local variables focus1,
    // focus2, and majorAxis
    ...
    PointGeometry p = new PointGeometry(x, y);
    return (focus1.distance(p) + focus2.distance(p))
        <= majorAxis;
}

```

比起本节中EllipseGeometry类的实现，这种方法有何优点和缺点。

4.25 你可能已经发现了，保存在域focus1和focus2中点有时并不是准确的真正的椭圆的焦点。原因在于有时真正的椭圆焦点的坐标并不是整数，而是实数，而保存焦点的PointGeometry对象只能保存整数坐标。一种解决方法是用Java的Point2D.Double类（在java.awt.geom包中）以浮点型的点坐标保存椭圆的焦

点 (`Point2D.Double` 是 `Point2D` 类中一个静态的嵌套类)。表达式：

```
new Point2D.Double(5.1, 6.2)
```

创建一个新的 `Point2D` 对象来表示点 (5.1, 6.2)。按照这种方法，`EllipseGeometry` 类的域 `focus1` 和 `focus2` 域应声明如下：

```
Point2D.Double focus1, focus2;
```

按照这种做法实现 `EllipseGeometry` 类。

[提示：表达的变化要求你修改原来的 `computeFoci` 和 `contains` 方法的实现。可以利用 `Point2D.Double` 对象的 `getX` 和 `getY` 方法来得到它的 `x` 和 `y` 的坐标。下面的方法：

```
// method of Point2D.Double class
public double distance(Point2D p)
```

可以用来计算点 `p` 到这个点的距离。]

4.26 编写一个图形应用程序：在一个椭圆内画 n 个随机的点，点的颜色也是随机的。程序有五个参数：

```
> java PaintPointsInEllipse n x y width height
```

椭圆的位置在 (x , y) 且宽度为 $width$ ，高度为 $height$ 。

[提示：随机点应该在椭圆外接矩形内部，但如果在椭圆外矩形内就要丢弃它。]

4.27 对于所有的圆，其周长和直径的比率都是固定的，这个比率被命名为 π ，它约等于 3.1415926535898。计算 π 的方法之一是：创建一个正方形，再创建一个内接在此正方形里的圆。若圆的半径是 r ，正方形每条边的长度就是 $2r$ 。在正方形内部产生很多随机点 (个数为 N)，记录下出现在圆内部的随机点的个数 M ，这样我们就得到了一个分数 $P=M/N$ ，它可以产生 π 的近似值： $\pi=4P$ 。

这种方法的原理在于 P 近似于圆 A_c 的面积 (πr^2) 和正方形 A_s 的面积 ($(2r)^2$) 之比。 $\pi=4P$ 是由下式推出的：

$$P \approx \frac{A_c}{A_s} = \frac{\pi r^2}{(2r)^2} = \frac{\pi r^2}{4r^2} = \frac{\pi}{4}$$

编写一个 `FindPi` 程序，带有一个长整数参数 N ，表示要产生的随机点的个数，输出近似于 π 的结果：

```
> java FindPi 100
rectangle: (0,0),10000,10000
ellipse: (0,0),10000,10000
pi = 3.16
> java FindPi 10000
rectangle: (0,0),10000,10000
ellipse: (0,0),10000,10000
pi = 3.1388
> java FindPi 1000000
rectangle: (0,0),10000,10000
ellipse: (0,0),10000,10000
pi = 3.141276
```

4.4.3 有理数

在这一部分，我们将讨论表示有理数的类。这个类的表达一致性约束会更复杂，也更

有趣。如果 a 和 b 都是整数，且 b 是不等于零，比值 a/b 为有理数， a 叫做分子而 b 叫做分母。下面是 Rational 类的类框架：

```
public class Rational {

    public Rational(long num, long denom)
        throws IllegalArgumentException
    // EFFECTS: If denom equals zero throws
    //   IllegalArgumentException; else initializes
    //   this to num/denom.

    public Rational(long num)
    // EFFECTS: Initializes this to the rational num.

    public Rational()
    // EFFECTS: Initializes this to the rational 0.

    public Rational add(Rational a)
    // EFFECTS: Returns the rational equal to this + a.

    public Rational multiply(Rational a)
    // EFFECTS: Returns the rational equal to this * a.

    public Rational subtract(Rational a)
    // EFFECTS: Returns the rational equal to this - a.

    public Rational divide(Rational a)
        throws IllegalArgumentException
    // EFFECTS: If a equals zero throws
    //   IllegalArgumentException; else returns the
    //   rational equal to this / a.

    public boolean equals(Object a)
    // EFFECTS: Returns true if a is a Rational and
    //   a denotes the same rational number as this;
    //   else returns false.

    public int compareTo(Object obj)
        throws ClassCastException
    // EFFECTS: If this and obj are incomparable throws
    //   ClassCastException; else returns a negative
    //   integer, zero, or a positive integer if this
    //   is less than, equal to, or greater than obj,
    //   respectively.

    public boolean isInteger()
    // EFFECTS: Returns true if this denotes an
    //   integer; else returns false.

    public long numerator()
    // EFFECTS: Returns the numerator.

    public long denominator()
    // EFFECTS: Returns the denominator.

    public String toString()
    // EFFECTS: Returns the "num/denom".

    public double toDouble()
    // EFFECTS: Returns a double approximating this.
}
```

观察这个类你会发现：没有一个方法会改变Rational对象的状态，即没有一个方法的描述包含修改语句，也没有一个方法改变有理数分子和分母。另外，`add`方法并不改变当前对象，它会产生一个新的Rational对象，如下所示：

```
Rational a = new Rational(2, 4);
Rational b = new Rational(2, 8);
Rational c = a.add(b);
System.out.println("a: " + a);    // a: 1/2
System.out.println("b: " + b);    // b: 1/4
System.out.println("c: " + c);    // c: 3/4
```

同样，`subtract`，`multiply`，`divide`等方法也不会改变Rational对象的状态。状态不能被客户改变的对象被称为是不可变的（immutable）。Rational对象就是这样的对象，它没有提供改变状态的公有增变器。相反，PointGeometry对象是可变的（mutable），因为它提供了改变其状态的公有增变器（例如，`setX`和`setY`方法）。

不可变对象类型（如Rational）应该定义一个相等（`equals`）方法。因为任何情况下，具有相同值的两个不可变的对象都是相等的。在对象创建时确定它的值之后，任何消息都不能改变不可变的对象的值，所以两个相等的不可变对象必然总是相等的。相反由于可改变对象（如PointGeometry对象）的状态可能会从一个状态变化到另一个状态，因此两个对象的相等是暂时的，它们可能会在任何时候被改变。

在实现Rational类之前，我们先演示该类的行为。下面的程序FindPiLiebniz执行时需要一个整型参数n，基于一个无穷数列前n项计算并打印出一个π的近似数。这个数列收敛于 $\pi/4$ ，它源于Leibniz：

$$1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \frac{1}{9} \dots$$

下面的程序利用了这个数列，参数n是要计算的项数：

```
public class FindPiLiebniz {
    public static void main(String[] args) {
        if (args.length != 1) {
            System.out.println("USAGE: FindPiLiebniz n");
            System.exit(0);
        }
        long n = Long.parseLong(args[0]);
        long sign = 1;
        long bottom = 1;
        Rational pi = new Rational(0);
        for (int i = 0; i < n; i++) {
            Rational term = new Rational(sign, bottom);
            pi = pi.add(term);
            sign = -sign;
            bottom += 2;
        }
        Rational answ = pi.multiply(new Rational(4));
        String msg = "pi = " + answ + " = " + answ.toDouble();
        System.out.println(msg);
    }
}
```

遗憾的是，Liebniz数列收敛得很慢。当你测试FindPiLiebniz程序时，你会发现当n的值超过22时，变量pi的值就会溢出。

下面讨论Rational类的存储结构。显然，应将有理数的分子和分母分别存储在两个长整形变量中：

```
// fields of Rational class
protected long num, denom;
```

由这两个域共同表示的有理数是 $num/denom$ 。但是这还不够，还要有限制Rational对象状态的条件。下面暂时离开现在所讨论的内容，先介绍一个概念。

两个正整数 a 和 b 的最大公约数（gcd）是能整除这两个数的最大整数。如果 n 能整除 a 和 b ，并且 n 是最大的能整除 a 和 b 的整数，则称 $\text{gcd}(a, b) = n$ 。12和18的最大公约数为6，3和9的最大公约数为3，4和7的最大公约数为1。如果两个正整数的最大公约数等于1，则这两个数是互质的（relatively prime）。4和7是互质，而12和18不是。

Rational类的表达一致性约束要用到最大公约数的概念。下面给出的一致性是针对存储结构（域num和denom）的：

- 最大公约数条件：如果num为非零，num和denom的绝对值应是互质的
- 零条件：如果num=0则denom==1
- 符号条件：denom>0

任何一个有理数都可以有无穷种不同的方法表示；例如 $1/2=2/4=18/36=(-200)/(-400)$ 。表达一致性约束的目的是保证每个不同的有理数只有一种表示方法。如果将所有Rational对象都化简成一种标准的表示形式，就可以保证任何两个代表同一个有理数的Rational对象会处在相同的状态。下面给出一些例子：

相等有理数 对象状态

1/2, 2/4, 18/36, (-200)/(-400)	→ num:1, denom:2
(-2)/3, 2/(-3), (-4)/6, 8/(-12)	→ num:-2, denom:3
0/1, 0/18, 0/(-36)	→ num:0, denom:1

采用一种标准的方法来表示有理数有很多好处，其中之一就是判断有理数相等很容易。两个Rational对象是相等的，当且仅当它们的分子和分母分别相等。另外，分子和分母的数量级都减小了，由此也减少了溢出的可能性。

有了存储结构，接下来我们将实现Rational类的方法。下面是三个构造器：

```
public Rational(long num, long den)
    throws IllegalArgumentException {
    if (denom == 0) throw new IllegalArgumentException();
    this.num = num;
    this.denom = den;
    normalize();
}

public Rational(long num) {
    this(num, 1);
}

public Rational() {
    this(0, 1);
}
```

两个参数的构造器先初始化域num和denom，然后调用normalize方法来建立其表达一致性约束。剩余的两个构造器不需要直接调用normalize方法，它们调用了两个参数的构造器。normalize方法应该定义如下：

```
// method of Rational class
protected void normalize() {
```

```

// MODIFIES: this
// EFFECTS: Establishes the representation
//   invariants such that this
//   denotes the rational num/denom.
long bigdivisor;
  // establish the sign condition
if (denom < 0) {
  num = -num;
  denom = -denom;
}
// establish the zero condition
if (num == 0)
  denom = 1;
  // establish the gcd condition
else {
  long tempnum = (num < 0) ? -num : num;
  bigdivisor = gcd(tempnum, denom);
  if (bigdivisor > 1) {
    num /= bigdivisor;
    denom /= bigdivisor;
  }
}
}

```

过程gcd计算两个整数的最大公约数，其定义如下：

```

// method of Rational class
protected long gcd(int a, int b) {
  // REQUIRES: a and b are positive.
  // EFFECTS: Returns the greatest common divisor
  //   of a and b.
  while (b > 0) {
    int rem = a % b;
    a = b;
    b = rem;
  }
  return a;
}

```

add方法只有一个有理数类型的参数，处理过程是将参数加到此有理数上并返回结果。同样，multiply方法也只有一个参数，将参数和此有理数相乘，然后返回结果。这两种方法都不改变参数和接收者的状态。下面是它们的用法的一个例子：

```

Rational a = new Rational(2,4);
Rational b = new Rational(2,8);
Rational c = a.add(b);
System.out.println("a: " + a); // a: 1/2
System.out.println("b: " + b); // b: 1/4
System.out.println("c: " + c); // c: 3/4
Rational d = c.multiply(a);
System.out.println("d: " + d); // d: 3/8

```

add和multiply方法使用了分数加法和乘法的标准规则，如下：

$$(a/b) + (c/d) = (ad + bc)/(bd)$$

$$(a/b) (c/d) = (ac)/(bd)$$

add过程和multiply过程的定义如下：

```

// methods of Rational class
public Rational add(Rational a) {

```

```

    long top = numerator() * a.denominator() +
               denominator() * a.numerator();
    long bot = denominator() * a.denominator();
    return new Rational(top, bot);
}

public Rational multiply(Rational a) {
    long top = numerator() * a.numerator();
    long bot = denominator() * a.denominator();
    return new Rational(top, bot);
}

```

可以看到add和multiply方法返回的Rational对象是通过调用构造器创建的，所以这个对象必定保持一致的状态：

```

Rational a = new Rational(2,4);
Rational b = new Rational(2,8);
Rational c = a.add(b);
// return new Rational(4*1 + 2*1, 2*4), same as
// return new Rational(6, 8); constructs
// a Rational with num==3 and denom==4

```

由于Rational对象都遵守标准的形式，所以比较两个对象是否相等可以通过分别比较它们的分子和分母是否相等来确定：

```

// method of Rational class
public boolean equals(Object obj) {
    if (obj instanceof Rational) {
        Rational a = (Rational) obj;
        return ((numerator() == a.numerator()) &&
                (denominator() == a.denominator()));
    }
    return false;
}

```

compareTo方法比较本对象和参数对象obj的大小，如果本对象小于、等于或大于参数obj，那么会分别返回负整数(-1)、零或正整数(1)：

```

// method of Rational class
public int compareTo(Object obj)
    throws ClassCastException {
    Rational a = (Rational) obj;
    long leftSide = numerator() * a.denominator();
    long rightSide = denominator() * a.numerator();
    if (leftSide < rightSide) return -1;
    else if (leftSide == rightSide) return 0;
    else return 1;
}

```

可以看到，compareTo方法与equals方法的含义是一致的。如果r和s都是Rational对象，表达式r.compareTo(s)返回0，当且仅当表达式r.equals(s)返回true。

isInteger方法判断此Rational对象是否为整数。该方法的实现利用了类的表达一致性约束，它保证了当Rational对象代表一个整数时，它的分母的值应该是1：

```

// method of Rational class
public boolean isInteger() {
    return denominator() == 1L;
}

```

总之，当Rational对象被创建时，其表达一致性约束也被建立，并且在对象的生存期内，由于对象的状态一直没有改变，其表达一致性约束也一直保持。实际上，所有的不可变对象都是这样的：构造器建立对象的表达一致性约束，由于它们的状态没有变化，所以在整个生存期都保持。与此相反，Ellipse对象是可变的，因此它的每个增变方法都必须保证遵守其表达一致性约束。对于可变对象，表达一致性约束应该被构造器建立，并被每一个改变状态的方法所遵守。

练习

4.28 完成Rational类的实现。一个非整数有理数表示成 $num/denom$ ，而一个整数有理数表示为 num ：

```
System.out.print("one-half: " + new Rational(3,6));
// one-half: 1/2
System.out.print("five: " + new Rational(10, 2));
// five: 5
```

4.29 计算 π 的近似值的另一种方法是这样的，根据Ernesto Cesaro以及《*Structure and Interpretation of Computer Program*》中的引用：两个随机整数互质的概率等于 $6/\pi^2$ 。编写一个程序利用这一原理计算 π 的近似值。利用本节的gcd方法反复测试两个随机数是否是互质的。（ a 和 b 是互质的当且仅当 $\text{gcd}(a, b)$ 等于1。）程序的long型参数 n 表示测试的随机数对的数量：

```
> java FindPiCesaro 1000
pi = 3.133682700398331

> java FindPiCesaro 1000000
pi = 3.1409239115514453
```

4.5 交互图形程序

到目前为止，我们已经讨论一些基于3.5节MyGraphicsProgram模板的图形程序。利用这个程序模板，可以编写出能够创建并显示常见图形的程序，但不允许用户在程序运行时交互操作。在这一节我们将介绍一个新的可交互程序模板——MyInteractiveProgram，它允许你定义一个控制器（controller）对象来管理用户和程序图形的交互。首先看一个基于MyInteractiveProgram模板的应用程序，之后再从这个程序中提取模板并进一步形式化。

4.5.1 随机点

PlaySplatterPoints程序用于在框架内容区内画随机点（Splattering Point）。用户可以通过命令来控制程序的图形效果，包括随机点的输出矩形区域、颜色、数目、显示或隐藏矩形。程序中两个重要的变量是当前颜色和当前矩形，用户通过它们来控制程序。当创建一个新的随机点时，它位于当前矩形内且为当前颜色。程序开始时，当前颜色是白色，当前矩形位于点（0, 0），宽和高都等于100。运行过程中用户可以改变这些值。程序提供以下命令来产生新的随机点以及更新当前矩形和当前颜色：

- `rectangle x y width height` 更新当前矩形到位置（ x, y ），宽度为 $width$ ，高度为 $height$ 。

- **show** 用白色轮廓线显示当前矩形。
- **hide** 隐藏当前矩形。
- **new *n*** 在当前矩形内产生*n*个随机点；这*n*个点都用当前颜色输出。
- **color *red green blue*** 当 $0 < \text{red}, \text{green}, \text{blue} \leq 255$ 时，更新当前颜色。
- **quit** 退出程序。

该程序由两个类组成。第一个类是**PlaySplatterPoints**，负责构造**frame**和**panel**对象，并创建和启动**controller**对象。这个类还实现了**paintComponent**方法。下面是这个类的完整定义：

```
public class PlaySplatterPoints extends ApplicationPanel {
    public PlaySplatterPoints() {
        setBackground(Color.black);
    }

    static PlaySplatterPointsController controller;

    public static void main(String[] args) {
        ApplicationPanel panel = new PlaySplatterPoints();
        ApplicationFrame frame =
            new ApplicationFrame("PlaySplatterPoints");
        frame.setPanel(panel);
        controller = new PlaySplatterPointsController(frame);
        frame.show();
        controller.start();
    }

    public void paintComponent(Graphics g) {
        super.paintComponent(g);
        Graphics2D g2 = (Graphics2D)g;
        g2.setRenderingHint(RenderingHints.KEY_ANTIALIASING,
                            RenderingHints.VALUE_ANTIALIAS_ON);
        Vector points = controller.points();
        Vector colors = controller.colors();
        for (int i = 0; i < points.size(); i++) {
            PointGeometry point = (PointGeometry)points.get(i);
            Color color = (Color)colors.get(i);
            g2.setPaint(color);
            g2.fill(point.shape());
        }
        if (controller.isVisible()) {
            RectangleGeometry rectangle=controller.rectangle();
            g2.setPaint(Color.white);
            g2.draw(rectangle.shape());
        }
    }
}
```

第二个类是**PlaySplatterPointsController**，它定义控制器的行为。这个类的对象要以独立的线程运行，类的定义要满足这一点。控制器和静态**main**方法（**PlaySplatterPoints**类的**main**方法）以相互独立的线程运行。这就允许控制器在**main**方法运行结束后还可以继续运行。**frame**也以自己的线程运行。

为了使**PlaySplatterPointsController**对象以其自己的线程运行，需将该类定义为**Thread**类的扩展类。用下面的两条语句，**PlaySplatterPoints**类的静态**main**方法创建新的**controller**对象并启动它运行：

```
controller = new PlaySplatterPointsController(frame);
controller.start();
```

下面是PlaySplatterPointsController类的实现：

```
class PlaySplatterPointsController extends Thread {

    protected ApplicationFrame frame;
    protected Vector points;
    protected Vector colors;
    protected RectangleGeometry rectangle;
    protected boolean isVisible;
    protected

        PlaySplatterPointsController(ApplicationFrame frame) {
            this.frame = frame;
            points = new Vector();
            colors = new Vector();
            isVisible = true;
            rectangle = new RectangleGeometry(0, 0, 100, 100);
        }

    protected Vector points() {
        return points;
    }

    protected Vector colors() {
        return colors;
    }

    protected RectangleGeometry rectangle() {
        return rectangle;
    }

    protected boolean isVisible() {
        return isVisible;
    }

    public void run() {
        ScanInput in = new ScanInput();
        Color currentColor = Color.white;
        RandomPoint rand = new RandomPoint();
        while (true) {
            try {
                System.out.print("? ");
                String s = in.readString();
                int x, y, w, h, r, g, b, n;
                switch (s.charAt(0)) {
                    case 'r': // rectangle x y width height
                        x = in.readInt();
                        y = in.readInt();
                        w = in.readInt();
                        h = in.readInt();
                        rectangle.setPosition(new PointGeometry(x, y));
                        rectangle.setWidth(w);
                        rectangle.setHeight(h);
                        break;
                    case 's': // show
                        isVisible = true;
                        break;
                    case 'h': // hide
                        isVisible = false;
                        break;
                }
            }
        }
    }
}
```

`PlaySplatterPointsController`类定义了五个域：

```
// fields of PlaySplatterPointsController class
protected ApplicationFrame frame;
protected Vector points;
protected Vector colors;
protected RectangleGeometry rectangle;
protected boolean isVisible;
```

`frame`域保存了对被控制框架的引用，控制器需要这一引用以便按照需要刷新框架。`points`域保存了已产生的所有随机点，`colors`域保存了这些随机点的对应颜色（`colors[i]`保存了点`Point[i]`的颜色）。`rectangle`域保存了当前矩形。`isVisible`域表示当前矩形是否在屏幕上可见。

为了提供对图形内容数据的访问，类 `PlaySplatterPointsController` 定义了 `points`、`colors`、`rectangle` 和 `isVisible` 方法来提供对其图形内容的访问（称这些方法为内容选择方法（content selector method））。方法 `paintComponent` 调用内容选择方法，以找出它要画什么。这四个方法共同定义了控制器的接口，便于 `PlaySplatterPoints` 类使用这些接口来显示图形。

`run`方法定义控制器的主要行为。控制器一启动`run`方法就被调用执行，并且运行在自己的线程中。`run`方法重复地读取和执行用户的命令，在每次重复的结尾，它都会发送`repaint`消息给框架，以确保显示是最新的。

更好地理解系统设计的一个方法就是研究场景。场景 (scenario) 模拟了外部代理 (比如用户) 和系统关键元素一系列消息顺序交互的过程。例如，在随机点程序中，用户输入 new 命令创建 n 个随机点时，一个场景就出现了。同样用户发布 rectangle 命令时，另一

个场景出现了。

UML提供了两种交互图描述场景：顺序图（sequence diagram）和协作图（collaboration diagram）。这两种图都描述了对象之间的动态交互关系，但是它们的侧重点不同：顺序图着重体现对象间消息传递的时间顺序，协作图着重体现交互对象间的静态链接关系。本书中使用顺序图（参见附录C获得有关顺序图的更多信息）。

顺序图有两个轴：水平轴上是表示不同的对象，垂直向下轴表示时间，时间自上而下。从垂直方向上看，顺序图由一组垂直列组成，每个列是参与特定场景的一个对象（列最上端的方框表示对象）。从每个对象方框中引出一条垂直虚线——对象的生命线；表示在某段时间内对象是存在的。对象间消息的传递在对象生命线间用带箭头的直线表示，箭头指向接收方。这样从上到下，消息是有次序地在对象之间传递，顺序地描述对象间事件的发展过程。有点抽象，最好看一个例子。图4-6描述了这样一个场景：用户用new n 命令创建 n 个新随机点，其中 n 是整数。在这个场景中，关键代理是用户、控制器对象（PaintSplatterPointsController的实例）、面板对象（PaintSplatterPoints的实例）和框架对象（ApplicationFrame的实例）。用户是外部代理（external agent），一个属于外界的元素；而其他三个属于系统自身的。

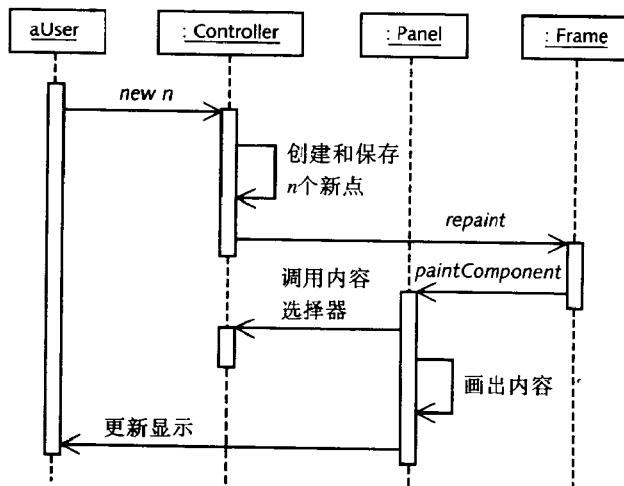


图4-6 创建 n 个随机点的顺序图

对于图4-6，有两个方面需要注意：第一，为了表示一个对象发送消息给它自己，可以从对象的生命线上画一条带箭头的线到它的自身；第二，顺序图模拟的是在某一层次上的抽象场景，意味着不可能描写那些很具体的交互过程。如在创建和保存新随机点的过程中，控制器对象重复地发送add消息给points和colors矢量对象，而这些交互过程层次太低，不能在这个顺序图中表达。事实上，points和colors矢量对象就没有出现在顺序图中。类似地，在框架收到repaint消息时会有种种交互产生，但对于这个场景，这些交互是属于细节上的，因而没有描述它们。

4.5.2 交互图形程序模板

下面从程序PlaySplatterPoints中抽取一个模板来作为交互程序的基础。该模板

由两个类组成：MyInteractiveProgram和MyInteractiveProgramController。在正式程序设计中，用程序名和控制器名分别替换模板中的这两个名字。MyInteractiveProgram类负责创建框架和面板、创建控制器并启动它。这个类也负责在面板上绘制图形。当然如果程序要带参数执行，也可以定义一个静态方法parseArgs（像在MyGraphicsProgram模版中那样）。下面是第一个类的形式：

```
public class MyInteractiveProgram
    extends ApplicationPanel {

    public MyInteractiveProgram() {
        ...
    }

    protected
        static MyInteractiveProgramController controller;

    public static void main(String[] args) {
        ApplicationPanel panel = new MyInteractiveProgram();
        ApplicationFrame frame =
            new ApplicationFrame("My Program");
        frame.setPanel(panel);
        controller =
            new MyInteractiveProgramController(frame);
        frame.show();
        controller.start();
    }

    // paints the contents of this panel
    public void paintComponent(Graphics g) {
        super.paintComponent(g);
        Graphics2D g2 = (Graphics2D)g;
        g2.setRenderingHint(RenderingHints.KEY_ANTIALIASING,
            RenderingHints.VALUE_ANTIALIAS_ON);
        ...
    }
}
```

第二个类是定义控制器的类，用来维护存储结构中保存的图形内容。它要提供内容访问方法，供paintComponent方法调用来查询显示的图形内容。由于控制器要以独立的线程运行，所以它要实现run方法，在控制器启动时调用。下面是这个类的形式：

```
class MyInteractiveProgramController extends Thread {

    // storage structure
    protected ApplicationFrame frame;
    // other fields specific to this application
    ...

    // constructor
    protected
        MyInteractiveProgramController(ApplicationFrame f) {
            this.frame = f;
            ...
        }

    // content selector methods required by
    // the paintComponent method
    ...
}
```

```
public void run() {
    ...
}
```

练习

4.30 请画出这样的场景的顺序图：在程序 `PlaySplatterPoints` 中，用户输入命令 `new n`，其中 `n` 是一个无效的整数。

4.31 在 `PlaySplatterPoints` 程序中增加下面两条命令：

- `clear` 删除迄今为止产生的所有随机点。
- `recolor` 用当前颜色刷新当前矩形内部的所有随机点，当前矩形外的随机点颜色不变。

4.32 根据 `MyInteractiveProgram` 模板，编写一个在框架内容区域画线段的交互程序 `PlayDrawLines`。用户输入 `line` 命令时，用当前颜色（初始为白色）画线。下面是程序支持的命令：

- `line x0 y0 x1 y1` 用当前颜色画一条从 (x_0, y_0) 到 (x_1, y_1) 的线。
- `color red green blue` 指定当前颜色，其中 $0 \leq red, green, blue \leq 255$ 。
- `quit` 退出程序。

4.33 修改上面程序 `PlayDrawLines`，增加一个初始值为 1 的 `stroke-width` 域，用来确定所画线的宽度：

- `stroke n` 更新 `stroke-width` 值为整型值 `n`。

4.34 在 `MyInteractiveProgram` 模板中，由于图形内容保存在控制器类中，控制器类必须提供内容选择方法，供 `MyInteractiveProgram` 的方法调用。这个问题可以用另一种实现方法解决：把控制器类作为 `panel` 类的内部类（inner class）定义，这样图形内容保存在属于 `panel` 类的作用域中，`panel` 类可以直接访问控制器中的数据。使用这种方法，修改 `PlaySplatterPoints` 类的实现。另外，以后要使用 `MyInteractiveProgram` 模板时，最好用这里描述的方法定义控制器类。

小结

面向对象程序设计的一个重要优点在于它支持软件元素的重用。重用的两个重要的机制是组合和继承。组合是一种类与类之间的整体与部分的关系，即一个类（组合体）的实例中包含另一个类（组件）的实例。每一个组件只属于它的组合体，并且它的生存期也受组合体的控制。由组合产生的复杂对象的结构是分层的：一个组合体是由若干个较简单的对象组成的，而这些对象又由一些更简单的对象组成，就这样继续分解下去，一直分解到最简单的（非组合）对象为止。聚集是另一种类与类之间的整体与部分关系。尽管 UML 并没有完全的规定出聚集的含义，聚集经常被应用于部分需要被多个整体所共享的情况下。

使用组合时，有时组件的数目是很大的，而且有时数目可能会随时间变化，相同类型的组合也可能有不同数目的组件。在这种情况下，通常要利用集合对象维护组件，这个对象集合提供查找元素、插入元素、删除元素等服务。这样的集合可以是数组、向量或

实现了 `java.util.Collection` 接口的类。组合体利用对象集合所提供的服务来管理它的组件，也可以把其中的一些服务当成自己的公有接口。

每个对象都有自己的记忆方式，又称为状态。状态的值保存在其对象的域里。对象的状态影响它的行为，如果是可变对象，其状态可能会变化。为了保持在一个合法的状态下，一个对象必须遵守表达一致性约束。对象建立时以及在两次调用其公有方法之间，表达一致性约束必须被满足。

第5章 继承

面向对象模型提供了两种主要的用已有类定义新类的机制：组合和继承。组合已经在第4章讨论过，这一章我们将讨论继承。使用继承，一个新类可以继承已有类的行为，然后可以修改这些行为，也可以定义新的行为，即新类可以修改它获得的某些行为，也可以定义已有类中没出现的或已有类指出但没有实现的新的行为。继承方式中，新类所描述的抽象和已有类所描述的相似，但却是不同的。

5.1节介绍继承的使用，又称继承的形式。5.2节到5.4节介绍这些继承的形式。5.5节讨论了多态性。5.6节把多态性应用到一些新类中，用来给几何图形加上外观表示。

5.1 继承的使用

使用继承我们可以定义一个新类作为已有类的扩展（extension）。相对于已有类，新类叫做子类（subclass或child class）；相对于新类，已有类称为超类（superclass）或父类（parent class）。一个子类又可以作为任意其他类的父类，如此不断构造就形成了继承层次结构（inheritance hierarchy）。图5-1就是这样一个类的继承层次结构图。在这个图中，类Boat是Watercraft的一个子类，Boat又是Motorboat，Paddleboat和Sailboat的父类。类Watercraft位于这个继承层次的根。

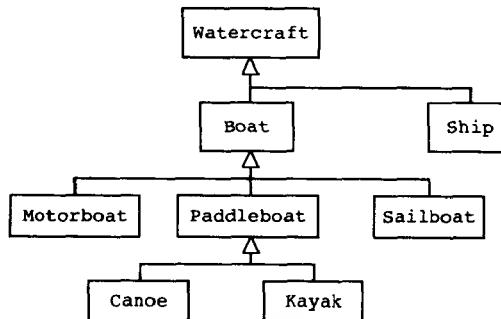


图5-1 一个继承的层次结构图

一个类B称为某一个类A的后代（descendant），如果B是A的子类或B是A的某个子类的一个后代。很显然，类A的后代就是在继承层次中以类A为根的那些类。假如类B是类A的一个后代，则类A称为类B的祖先（ancestor）。在图5-1中除了类Watercraft自己以外，出现的每一个类都是类Watercraft的后代。类Kayak的祖先是类Paddleboat、Boat和Watercraft。这个类的祖先组成从这个类到根类的一个类链。

一个继承层次的根类作为类本身和它所有的后代的父型（supertype）。根类的后代和根类本身都是根类的子型（subtype）。例如，在图5-1中，所有的类都是类Watercraft的子型，而类Paddleboat的子型仅包括类Canoe、Kayak和Paddleboat本身。一个父型规定了它的每一个子型都提供的接口，类Watercraft的每一个子型至少提供了它规定的

行为。类型家族的建立同样是使用Java的接口机制，这将在5.4节中介绍。

图5-1表明（真正的案例），一个类可以有任意数目的子类。在Java中除了类java.lang.Object没有父类外，每一个类都有确定的父类。Object类位于全局继承层次的根部，所有的类都属于这个继承层次。一个类层次结构图仅表示那些和考察系统紧密相关的类，像类Object和其他一些和特定子系统不紧密相关的类，通常是不会在图中详细描述的。

你或许熟悉通过继承定义一个新类的语法。如果A是一个类，一个新的子类B可以这样定义：

```
class B extends A {
    // fields and methods of B
    ...
}
```

如果extends子句省略掉，那么这个新类就成为类Object的子类。下面的类C就是从类Object扩展出来的：

```
class C {
    // fields and methods of C
    ...
}
```

继承是一种重用机制：一个子类可以重用它的父类，它重用的是父类的域和方法，准确地说，每一个子类获得的是父类的非私有的域和方法。为保持这种家族关系的延续，我们称子类从父类继承（inherit）了这些元素。

如果一个子类没有提供它自己的任何定义，它的结构和行为和它的父类没有区别，这样做几乎没有任何意义。继承更多地是用来定义一个新类，新类扩展、修改和定义父类的结构和行为，亦即定义新的方法、覆盖继承的方法和实现父类仅定义但没有实现的方法。新子类定义了和父类类似的新类型，但也有些区别。试看一个简短的例子，其中类Waiter扩展子类Employee。这是一个继承最自然的使用，因为在饭店中，服务生是雇员的一种：

```
public class Employee {
    public void greeting(String name) {
        System.out.println("Welcome to Eat and Run, "+name+".");
    }
    public void farewell(String name) {
        System.out.println("Please come back soon, "+name+"!");
    }
}
class Waiter extends Employee {
    public void greeting(String name) {
        System.out.println("Can I take your order, "+name+"?");
    }
    public void recommendation() {
        System.out.println("I'd stick to the pizza.");
    }
}
```

类Employee定义了两个方法greeting和farewell，它们被Employee的子类Waiter继承。除了继承两个已有的方法外，通过定义一个新的方法recommendation，Waiter类添加了一个新的行为（和一般的雇员不同，服务生是为顾客取送食物的）。另外类Waiter重新定义了greeting方法，这是一个覆盖的例子，因为Waiter的greeting方法和父类的greeting方法有同样的形式：

```
public void greeting(String name)
```

类Waiter继承了farewell方法，而且没有改变它（没有在Waiter类定义中出现farewell方法）；一般的employee和特别的waiter用一样的方式与客人说再见。下面是一个使用这些类的简短的程序例子：

```
class TryEmployeeAndWaiter {
    public static void main(String[] args) {
        String name = "Elisa";
        if (args.length > 0)
            name = args[0];
        Employee e = new Employee();
        Waiter w = new Waiter();
        e.greeting(name);    // Welcome to Eat and Run, Elisa.
        w.greeting(name);    // Can I take your order, Elisa?
        w.recommendation();  // I'd stick to the pizza.
        w.farewell(name);   // Please come back soon, Elisa!
        e.farewell(name);   // Please come back soon, Elisa!
    }
}
```

继承通常用来定义一个类，这个类是它的父类一个特殊的类型。一个子类和它的父类遵循“是(is-a)”关系：子类的每一个实例都是它的父类的一个实例。这可以从图5-1的类得到证实。较特殊化的子类可以提供它的父类所有的行为，也可以增加它自己的行为。再来看Waiter和Employee类，每一个服务生都是一种雇员，并且每一个服务生都有和一般的雇员相同的行为；另外，和一般的雇员不同，服务生可以给出一些建议行为。

不幸的是，恰当地使用继承不会像发现两个概念是否具有“是”关系那么简单。当两个概念符合“是”关系时，它们的类之间不一定会有继承关系，有时候是不可能具有继承关系。举例来说，一个概念可能会含有多个意思，但是它的类只允许从一个父类扩展（一个助教既是大学雇员又是学生，但是这个TeachingAssistant类可以只有一个父类）。此外，“是”关系有时候通过接口实现会更好，接口可以用来表达更普遍的概念。由于这种复杂性，所以使用继承前要考虑清楚使用它的原因。

在第3章，我们讲述了如何利用数据抽象和封装来创建接口和实现它。公有接口知道数据类型，而它的实现却隐藏起来，因为它的客户没有必要知道。考虑继承时，区分接口和实现之间的差别是很重要的。一个子类扩展它的父类是为了继承父类的接口，还是继承它的实现，或者是两者都继承呢？答案取决于为什么使用继承。共有三种主要的使用继承的原因，这有时候也称为继承的形式：

- **扩展继承 (Inheritance for extension)**: 子类扩展父类是为了定义新的行为作为补充。子类定义新的方法和（或）域，但是没有覆盖它继承的方法。子类增加的新行为没有在它的父类中出现也不适用于父类。这种情况下，父类的接口和实现都被继承了。
- **特化继承 (Inheritance for specialization)**: 子类扩展父类是为了修改父类中的一些行为，同时保持它继承的其他行为不变。这种情况下，子类继承了父类的接口和一部分实现，而覆盖了其他的实现。
- **说明继承 (Inheritance for specification)**: 先要假设父类定义了一些抽象方法。抽象方法是被父类声明了但没有实现的方法，代表了行为的说明。说明继承的理想形式是：父类是接口或者是所有方法都没有实现的抽象类，子类只继承完整的接口。而在另

一种混合形式中，父类是已经实现了部分方法的抽象类，子类继承了完整接口和部分方法的实现。在这两种继承形式中，子类都是按父类提供的说明实现抽象行为。

在实际情况中，一个子类继承它的父类常常不是因为上面的一种原因，而是两种或三种原因都有。在前面的例子中，`Waiter`类扩展了`Employee`，就是因为前两种原因：`Waiter`类既添加了一个新的行为（`recommendation`方法），又覆盖了它继承的行为（`greeting`方法）。

为了理解继承，我们将分别讲解三种继承形式。这是下面三节内容的目标。

5.2 扩展继承

当使用扩展继承时，一个子类定义新的域和方法作为对已继承行为的补充。在这一节，我们将使用这种继承形式来开发一个可以进行缩放和旋转的点类。点变换功能是由父类`PointGeometry`提供的行为和另外增加的几个新方法完成。我们还会定义一个描述无穷直线的类。先看一个简单的扩展继承的例子。

5.2.1 N步计数器

一个N步计数器（n-step counter）是一个计数器，它的值是按固定的整数增加或减少，这个整数叫做步（step）。一个`NstepCounter`对象的步在它被构造时初始化；它用`inc`方法增加计数器，每次增加值为步；用`dec`方法减少计数器，每次减少值为步；用`step`方法来获得步的值；用`value`方法获得它当前的值。下面是这个类的类定义：

```
public class NStepCounter {

    protected int step, value;

    public NStepCounter(int step) {
        // EFFECTS: Initializes this counter to
        // value zero and given step.
        this.step = step;
        this.value = 0;
    }

    public NStepCounter() {
        // EFFECTS: Initializes this counter's value
        // to zero and step to 1.
        this(1);
    }

    public void inc() {
        // MODIFIES: this
        // EFFECTS: Increments value by step.
        value += step;
    }

    public void dec() {
        // MODIFIES: this
        // EFFECTS: Decrements value by step.
        value -= step;
    }

    public int step() {
        // EFFECTS: Returns this counter's step.
        return step;
    }
}
```

```

    }

    public int value() {
        // EFFECTS: Returns this counter's value.
        return value;
    }
}

```

利用扩展继承方法定义一个新类ClearableCounter，除了增加一个clear方法来重置计数器值为0，其他行为和NstepCounter相同。下面是这个类的定义：

```

public class ClearableCounter extends NStepCounter {

    public ClearableCounter(int step) {
        // EFFECTS: Initializes this counter
        // to value zero and given step.
        super(step);
    }

    public ClearableCounter() {
        // EFFECTS: Initializes this counter's value
        // to zero and step to 1.
        this(1);
    }

    public void clear() {
        // MODIFIES: this
        // EFFECTS: Resets this counter's value to zero.
        value = 0;
    }
}

```

ClearableCounter类增加了新的方法clear，但没有重新定义它继承的方法。注意ClearableCounter类还继承了父类的value域，并在它的clear方法中使用。

这两种类型的计数器中，它们的value方法都返回从计数器被清零后通过连续调用inc和dec方法得到的总和。它们的区别在于NstepCounters只在构造时被清零一次，而类ClearableCounter在构造时和调用clear方法时都可清零。

练习

5.1 因为ClearableCounter类没有提供step的获得者和设置者方法，所以step不是类的特性。定义一个step是特性的新类VariableNStepCounter，该类应该扩展类ClearableCounter并增加下面的方法：

```

public void setStep(int newStep)
    // MODIFIES: this
    // EFFECTS: Changes this counter's step to newStep.

public int getStep()
    // EFFECTS: Returns this counter's step.

```

5.2 编写一个应用程序来测试NstepCounter类。可以通过下面命令行形式执行你的程序：

```
>java TryNStepCounter step n1 n2 ...
```

该程序创建了一个新的N步计数器count，它的步是通过程序的第一个参数确定

的。然后按下面的叙述来处理剩下的参数：增加count计数器 n_1 次并打印它的状态，然后减小count计数器 n_2 次并打印它的状态，等等。它处理程序的参数是从左到右，如果*i*是奇数，增加计数器 n_i 次，如果*i*是偶数，减少计数器 n_i 次。下面是一个例子：

```
> java TryNStepCounter 2 6 3 9 1
step = 2
6 increments: value = 12
3 decrements: value = 6
9 increments: value = 24
1 decrements: value = 22
```

5.2.2 可变换点

几何变换 (geometric transformation)，简称变换 (transformation)，在计算机绘图中有很多用途。几何变换可用来建立模型；几何变换也用来生成三维图形；几何变换还用在动画中。例如，描绘虚拟照相机的移动和场景事物的移动和变化，例如，几何形状、光线和纹理。这里主要考虑如何使用几何变换来完成修改图形对象的形状、位置和方位。尽管几何变换可适用于任意维数空间，这里只用二维变换变换平面上的点。

基本的变换方法是平移 (translation)、(按比例) 缩放 (scaling)、旋转 (rotation)。平移是沿一条直线把一个对象从一个位置移到另一个位置，而不改变对象的方位或形状。在图5-2中，一个点 $p = (4, 0)$ 沿x轴移动1，沿y轴移动4，它的新位置是 $p = (4, 0) + (1, 4) = (4+1, 0+4) = (5, 4)$ 。

缩放不仅改变对象的尺寸，还改变它的位置。如果x轴和y轴上的缩放比例因子相同，就得到均衡缩放 (uniform scaling)；否则，为非均衡缩放 (nonuniform scaling) 或差异缩放 (differential scaling)。平面上发生缩放的点，称为抛锚点 (anchor point)。抛锚点周围的区域在缩放操作的影响下伸展和收缩：如果比例因子大于1，空间就是伸展；如果比例因子小于1，空间就被收缩。如果一个被缩放对象是一个点，则经过操作后这个点的大小 (它是无限小的) 没有影响，但是它到抛锚点的距离是有变的。在图5-2中，一个点 $q = (3, 1)$ ，缩放是沿x轴 $sfx=2$ ，沿y轴 $sfy=3$ ，则它的新的位置是 $q' = (3*2, 1*3) = (6, 3)$ 。

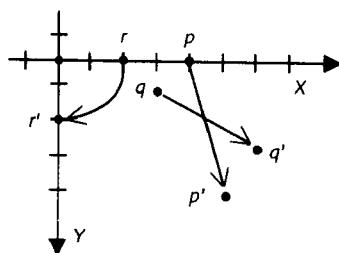


图5-2 点变换图

旋转将一个对象沿一个圆形轨迹重新定位，这个圆形轨迹的中心是抛锚点。旋转操作用一个角度 θ 表示旋转的角度。在Java的默认的坐标系下，如果 θ 是正数，那么旋转是沿顺时针方向的；如果 θ 是负数，旋转是沿逆时针方向的。在图5-2中，一个点 $r=(2, 0)$ 沿原点旋转 $\theta=90^\circ$ 到另一个位置 $r'=(0, 2)$ 。

TransformablePointGeometry类举例说明了纯粹的扩展继承——它定义了新的方

法而且没有覆盖任何继承的方法。下面的框架列出了这个类引人的新方法：

```

public class TransformablePointGeometry
    extends PointGeometry {

    public TransformablePointGeometry (int x, int y)
        // EFFECTS: Constructs a new transformable point
        // at (x,y).

    public TransformablePointGeometry (PointGeometry p)
        throws NullPointerException
        // EFFECTS: If p is null throws NullPointerException;
        // else constructs a new transformable point
        // at position p.

    public TransformablePointGeometry ()
        // EFFECTS: Constructs a new transformable point
        // at (0,0).

    public void rotate(double theta)
        // MODIFIES: this
        // EFFECTS: Rotates this point by theta degrees
        // around the origin.

    public void rotate(double theta, PointGeometry anchor)
        throws NullPointerException
        // MODIFIES: this
        // EFFECTS: If anchor is null throws
        // NullPointerException; else rotates this point
        // by theta degrees around anchor.

    public void scale(double sfx, double sfy)
        // MODIFIES: this
        // EFFECTS: Scales this point by scale factor sfx
        // along the x axis and by sfy along the y axis
        // relative to the origin.

    public void scale(double sf)
        // MODIFIES: this
        // EFFECTS: Scales this point uniformly by scale
        // factor sf, relative to the origin.

    public void scale(double sfx, double sfy,
                     PointGeometry anchor)
        throws NullPointerException
        // MODIFIES: this
        // EFFECTS: If anchor is null throws
        // NullPointerException; else scales this point
        // by scale factor sfx along the x axis and by
        // sfy along the y axis, relative to anchor.

    public void scale(double sf, PointGeometry anchor)
        throws NullPointerException
        // MODIFIES: this
        // EFFECTS: If anchor is null throws
        // NullPointerException; else scales this point
        // by scale factor sf, relative to anchor.
}

```

接下来的程序演示使用这个类的行为，斜体的注释为程序执行的结果：

```
public class TryTransformablePointGeometry {
```

```

public static void main(String[] args) {
    PointGeometry origin = new PointGeometry();
    TransformablePointGeometry p =
        new TransformablePointGeometry(3, 4);
    System.out.println("p: " + p); // p: (3,4)
    System.out.println("distance: "+p.distance(origin));
                                // distance: 5.0
    p.translate(1, -4);
    System.out.println("p: " + p); // p: (4,0)
    p.rotate(90);
    System.out.println("p: " + p); // p: (0,4)
    p.scale(2);
    System.out.println("p: " + p); // p: (0,8)
    p.scale(0.25);
    System.out.println("p: " + p); // p: (0,2)
    p.rotate(180, new PointGeometry(1, 2));
    System.out.println("p: " + p); // p: (2,2)
}
}

```

观察上面程序可知，TransformablePointGeometry继承了父类PointGeometry行为。举例来说，上面测试程序调用的distance方法是从父类继承来的，类似地当p被转化成字符串时，隐含调用从父类继承的toString方法。接下来看该类的详细定义：

```

public class TransformablePointGeometry
    extends PointGeometry {

    public TransformablePointGeometry(int x, int y) {
        super(x, y);
    }

    public TransformablePointGeometry(PointGeometry p)
        throws NullPointerException {
        this(p.getX(), p.getY());
    }

    public TransformablePointGeometry() {
        this(0, 0);
    }

    public void rotate(double degrees) {
        double radians = Math.toRadians(degrees);
        double cos = Math.cos(radians);
        double sin = Math.sin(radians);
        int newX =
            (int) Math.round(cos * getX() - sin * getY());
        int newY =
            (int) Math.round(sin * getX() + cos * getY());
        setX(newX);
        setY(newY);
    }

    public void rotate(double degrees, PointGeometry anchor)
        throws NullPointerException {
        translate(-anchor.getX(), -anchor.getY());
        rotate(degrees);
        translate(anchor.getX(), anchor.getY());
    }

    public void scale(double sfx, double sfy) {
        setX((int) Math.round(sfx * getX()));
        setY((int) Math.round(sfy * getY()));
    }
}

```

```

    }

    public void scale(double sf) {
        scale(sf, sf);
    }

    public void scale(double sfx, double sfy,
                      PointGeometry anchor)
        throws NullPointerException {
        translate(-anchor.getX(), -anchor.getY());
        scale(sfx, sfy);
        translate(anchor.getX(), anchor.getY());
    }

    public void scale(double sf, PointGeometry anchor)
        throws NullPointerException {
        scale(sf, sf, anchor);
    }
}

```

先来看一个参数的rotate方法，它把点 $p=(x, y)$ 围绕原点旋转 θ 角度，产生新的点 $p'=(x', y')$ 可以用三角法来表示：

$$x' = x \cos \theta - y \sin \theta \quad y' = x \sin \theta + y \cos \theta$$

旋转方向从正 x 轴到正 y 轴。因为Java的三角函数需要输入弧度，所以需要用Math.toRadians方法把角度转换成弧度。

两个参数的rotate方法围绕输入点anchor旋转。实现时要使它围绕原点旋转：首先把点平移- $anchor$ ，然后围绕原点旋转，最后平移 $anchor$ 回去。（- $anchor$ 代表点(- $anchor.getX()$, - $anchor.getY()$)。点的缩放做法类似：首先平移- $anchor$ ，然后把它关于原点缩放，最后平移 $anchor$ 回去。

下面看绘图程序PaintCirclePoints，它使用了TransformablePointGeometry类。这个程序在一个半径确定的不可见的圆周上均匀放置 n 个点，我们把这个圆叫做导向圆（guide circle）。这些点被绘制成半径为2、填充随机颜色的小圆。

程序使用3.5节的MyGraphicsProgram模板，用PaintCirclePoints类替代MyGraphicsProgram类。除了需要定义makeContent和paintComponent方法外，还要定义类所需要的静态域和静态方法parseArgs。

PaintCirclePoints类定义了两个静态域：

```
// static fields of PaintCirclePoints class
protected static int nbrPoints = 20, radius = 100;
```

`nbrPoints`为均匀放置点数，`radius`为导向圆半径。它们在定义时给出了初始默认值，但可以通过程序带参数执行修改。程序运行命令如下：

```
> java PaintCirclePoints [n [radius]]
```

如果没有参数输入，上面两个域使用默认值；如果带有一个或两个参数，默认值就会被相应替换。看下面处理参数方法parseArgs就可明白：

```
// static method of PaintCirclePoints class
public static void parseArgs(String[] args) {
    if (args.length > 2) {
        String s="USAGE: java PaintCirclePoints [n [radius]]";
        System.out.println(s);
    }
}
```

```

        System.exit(0);
    }
    if (args.length > 0)
        nbrPoints = Integer.parseInt(args[0]);
    if (args.length > 1)
        radius = Integer.parseInt(args[1]);
}

```

类PaintCirclePoint的实例域保存一个可变换点数组：

```

// field of PaintCirclePoints class
protected TransformablePointGeometry[] points;

```

makeContent方法中创建nbrPoints个沿导向圆定位的可变换点数组并把它们逐个赋给points域。下面是这个方法的实现：

```

// method of PaintCirclePoints class
public void makeContent() {
    double deltaDegrees = 360.0 / nbrPoints;
    double curDegrees = 0.0;
    points = new TransformablePointGeometry[nbrPoints];
    // create point and position it at (radius, 0)
    TransformablePointGeometry eastPoint =
        new TransformablePointGeometry();
    eastPoint.translate(radius, 0);
    // fill array points with copies of eastPoint
    // positioned along the guide circle
    for (int i = 0; i < nbrPoints; i++) {
        TransformablePointGeometry p =
            new TransformablePointGeometry(eastPoint);
        p.rotate(curDegrees);
        points[i] = p;
        curDegrees += deltaDegrees;
    }
}

```

makeContent方法工作过程如下：先创建一个可变换点命名为eastPoint，然后把这个点平移到位置 (radius, 0)，导向圆最“东边”的点。然后反复沿着导向圆的圆周上放置nbrPoints个点。每一次先构造eastPoint点的一个拷贝，然后把这个拷贝沿着导向圆旋转curDegrees（不断增加）角度，并把它存储在数组points中。

PaintComponent方法使用随机颜色画出保存在points数组中的点。这个方法定义如下：

```

// method of PaintCirclePoints class
public void paintComponent(Graphics g) {
    super.paintComponent(g);
    Graphics2D g2 = (Graphics2D)g;
    // center the rendering context g2 in the frame
    Dimension d = getFrame().getContentSize();
    g2.translate((int)(d.width/2), (int)(d.height/2));
    // paint each point with a random color
    RandomColor rndClr = new RandomColor();
    for (int i = 0; i < points.length; i++) {
        g2.setColor(rndClr.nextColor());
        PointGeometry p = points[i];
        g2.fill(p.shape());
    }
}

```

每一个绘图环境的默认坐标系和绘图表面的坐标系是一致的（原点在左上角， x 轴向右增加， y 轴向下增加）。在 `paintComponent` 方法中，为了导向圆能定位在框架中部，`Graphics2D` 对象 `g2` 被发送了 `translate` 消息。把绘图环境 `g2` 的坐标系平移到屏幕中部：

```
Dimension d = getFrame().getContentSize();
g2.translate((int)(d.width/2), (int)(d.height/2));
```

关于这部分，3.4节中有更详细的说明。为了画出每个点，`paintComponent` 方法遍历了 `points` 数组。对于每一个点 `p`，先设 `g2` 为一种新的随机颜色，然后用 `fill` 消息填充点。注意，框架每次刷新时，点的颜色都可能会发生变化。在本章后面的内容中，我们会扩展类来实现把外观（比如说填充颜色）和几何图形联系起来。

练习

5.3 类 `TransformablePointGeometry` 的实现过程中从没有使用过从父类 `PointGeometry` 继承的域 `x` 和 `y`，而是通过 `getX` 和 `getY` 方法访问 `x` 和 `y`，通过 `setX` 和 `setY` 来改变它们的值。也就是说，子类应尽量避免直接使用继承域，而是通过使用它继承的方法来访问和设置域。这样做的优点是什么？缺点是什么？

5.4 下面是 `PaintCirclePoints` 类的 `makeContent` 方法的另一个版本：

```
// method of PaintCirclePoints class: version 2(faulty)
public void makeContent() {
    double deltaDegrees = 360.0 / nbrPoints;
    points = new TransformablePointGeometry[nbrPoints];
    TransformablePointGeometry point =
        new TransformablePointGeometry();
    point.translate(radius, 0);
    for (int i = 0; i < nbrPoints; i++) {
        points[i] = point;
        point = new TransformablePointGeometry(point);
        point.rotate(deltaDegrees);
    }
}
```

解释这个版本的实现思想。为什么这个方法产生圆轨迹上点时经常会失败？试着找出原因。

5.5 编写一个应用程序 `PaintPointGrid`，它在不可见的栅格的交叉点来画随机颜色的点。程序有两个整型参数来声明连续的垂直的栅格线间的距离 (m) 和连续的水平栅格线间的距离 (n)：

```
> java PaintGridPoints m n
```

假设框架的内容区宽为 w 高为 h ：

```
// method makeContent of PaintPointGrid class
// (code fragment)
Dimension d = getFrame().getContentSize();
int w = d.width;
int h = d.height;
```

每一个点应该放在位置 (x, y) , $x = i \cdot m$, $y = j \cdot n$, 当 $0 \leq i < [w/m]$ 且 $0 \leq j < [h/n]$ 。

5.6 定义一个类 `ComparableAttribute` 作为 `Attribute` 类（练习 3.4）的子类。

`ComparableAttribute` 类定义一个新的方法，用来完成按照字母顺序比较名字的大小：

```

public int compareTo(Object obj)
    throws NullPointerException, ClassCastException
// EFFECTS: If obj is null throws
// NullPointerException; else if obj is not
// comparable throws ClassCastException;
// else returns a negative integer, zero, or
// positive integer if this attribute is less
// than, equal to, or greater than obj,
// under lexicographic ordering on name.

```

这样ComparableAttribute就可以像整数（用小于号比较）或者字符串（String）类的compareTo方法一样，进行线性比较排序了。接下来的代码段举例说明了compareTo方法的行为：

```

ComparableAttribute apple, grape;
apple = new ComparableAttribute("apple", "red");
grape = new ComparableAttribute("grape", "green");
apple.compareTo(grape);    // returns -1
grape.compareTo(apple);    // returns 1
apple.compareTo(apple);    // returns 0

```

5.2.3 直线

在这一节，我们将开发一个在平面上画直线的类。直线是可以向两个方向无限延伸的。一条直线是由通过这条线的任意一对不同的点 p_0 和 p_1 确定的，这两个点决定了这条直线和它的方向——这条线从点 p_0 指向 p_1 。由于直线是有方向的，所以依据它可以把平面上所有点分成三部分：任何点都位于直线的左边、直线的右边或是直线上。在图5-3中，直线的方向用开放箭头表示。

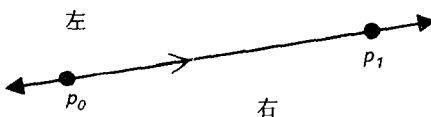


图5-3 平面上的点用一条直线分类

表示直线的类LineGeometry是从LineSegmentGeometry类扩展而来（参见练习3.3）。从行为方面来看，直线继承了线段的行为并增加了一个新的行为：用它可以把平面上的点分类。这里用直线作为扩展继承的一个例子。下面是这个类的类框架，按照惯例，这个框架仅仅说明新增的或覆盖的方法：

```

public class LineGeometry extends LineSegmentGeometry {

    public LineGeometry(int x0, int y0, int x1, int y1)
        throws IllegalArgumentException
    // EFFECTS: If (x0,y0) is equal to (x1,y1) throws
    // IllegalArgumentException; else constructs the
    // line (x0,y0)<-->(x1,y1).

    public LineGeometry(PointGeometry p0, PointGeometry p1)
        throws NullPointerException, IllegalArgumentException
    // EFFECTS: If p0 or p1 are null throws
    // NullPointerException; else if p0 equals p1
    // throws IllegalArgumentException;
    // else constructs the line p0<-->p1.
}

```

```

// values returned by point classification methods
public static final int LEFT = 0, ON = 1, RIGHT = 2;
public int classifyPoint(int x, int y)
    // EFFECTS: Returns LEFT, ON, or RIGHT if (x,y) lies
    // to the left of, on, or to the right of this line

public int classifyPoint(PointGeometry p)
    throws NullPointerException
    // EFFECTS: If p is null throws NullPointerException;
    // else returns LEFT, ON, or RIGHT if (x,y) lies to
    // the left of, on, or to the right of this line.

public String toString()
    // EFFECTS: Returns "p0<-->p1" where p0 and p1
    // determine this line.
}

```

接着来实现这个类。由于这里的直线和线段内部表示是一样的，所以没有必要来扩展存储结构。构造器定义如下：

```

public LineGeometry(int x0, int y0, int x1, int y1)
    throws IllegalArgumentException {
    super(x0, y0, x1, y1);
    if (getP0().equals(getP1()))
        throw new IllegalArgumentException();
}

public LineGeometry(PointGeometry p0, PointGeometry p1)
    throws NullPointerException, IllegalArgumentException {
    this(p0.getX(), p0.getY(), p1.getX(), p1.getY());
}

```

`classifyPoint`方法的参数是点 p 的 x 和 y 坐标，返回相对于直线的位置标识。下面是实现：

```

// static field of LineGeometry class
public static final int LEFT = 0, ON = 1, RIGHT = 2;

// method of LineGeometry class
public int classifyPoint(int x, int y) {
    int ax = getP1().getX() - getP0().getX(); // va=(ax,ay)
    int ay = getP1().getY() - getP0().getY();
    int bx = x - getP0().getX(); // vb=(bx,by)
    int by = y - getP0().getY();
    int res = ax * by - bx * ay;
    if (res < 0) return LEFT;
    else if (res > 0) return RIGHT;
    else return ON;
}

```

使用线性代数算法来实现`classifyPoint`方法。当 $p(x, y)$ 是一个输入点时，构造两个向量 $v_a = p_1 - p_0$ 和 $v_b = p - p_0$ 。然后它计算 $|v_a v_b|$ 并存储在本地变量`res`中，它等于以 p_0, p_1, p 为顶点的三角形的面积的两倍。`res`的符号决定了 p 点是位于当前直线的左边、右边还是直线上。

一个参数的`classifyPoint`方法是由两个参数的`classifyPoint`方法实现的：

```
// method of LineGeometry class
```

```

public int classifyPoint(PointGeometry p)
    throws NullPointerException {
    return classifyPoint(p.getX(), p.getY());
}

```

LineGeometry类覆盖了父类的toString方法，使直线有不同于线段的字符串描述符：

```

// method of LineGeometry class
public String toString() {
    return getP0() + "<-->" + getP1();
}

```

如果这里不覆盖toString方法的定义，则LineGeometry类是纯粹的扩展继承。

练习

5.7 编写一个图形应用程序，画一条直线L和n个随机点，每个随机点的颜色是由它和这条线的关系决定的。这个程序调用时有5个整型的参数：

```
> java PaintClassifyPoints n x0 y0 x1 y1
```

两个不同的点 (x_0, y_0) 和 (x_1, y_1) 确定了直线L。n个随机点中，每一个点都被分别的涂上红色、绿色或蓝色，这决定于它是在线L的左边、在线上还是在线的右边。

5.8 一条直线的表示不是惟一的：任何给定的直线都包含多种不同的表示。举例来说，正的x轴也可以用一对点 $p_0 = (0, 0)$ 和 $p_1 = (1, 0)$ 来表示或用另外一对 $p_0 = (-12, 0)$ 和 $p_1 = (6, 0)$ 表示。实际上，任何一对点 $p_0 = (x_0, 0)$ 和 $p_1 = (x_1, 0)$ 当 $x_0 < x_1$ 时都可以表示x轴。

当两条直线共线并且有同样的方向时，我们说这两个LineGeometry对象是相等的。因为直线的表示不是惟一的，所以不能用判断表示它们的点是否相等来测试。

实现方法equals来判断两条直线对象是否相等：

```

// method of LineGeometry class
public boolean equals(Object a)

```

把该方法添加到本节定义的LineGeometry类中。

equals方法可以按三步执行：

1) 如果a不是LineGeometry类的实例，返回false。

2) 检查这两条线是否共线。选择这条线上的两个点 p_0 和 p_1 ，把它们按和线a的关系分类。如果 p_0 或 p_1 不在线a上，equals返回false。否则，这两条线是共线的。

3) 检查这两条线（已经知道是在同一条直线上）是否有相同的方向。取一个不在线上的点 q ，如果 q 位于两条线的同一边，equals返回true，否则返回false。

编写一个短程序来测试你的equals过程的实现。

5.9 现在来看，类LineGeometry从它的父类继承了shape方法。这样具有同样的 p_0 和 p_1 的一条直线和一条线段的shape方法的形状效果是同样的，画出时为同一条线段。我们希望用一条向两个方向延伸的直线来描写直线，这就需要实现你的LineGeometry类中的shape方法。shape方法定义了两个点 q_0 和 q_1 ，它们都在线上，然后返回这一形状：

```

new Line2D.Float(q0.getX(), q0.getY(),
    q1.getX(), q1.getY());

```

q_1 在 p_1 外边，离 p_1 较远，从 p_0 到 q_1 的距离是 p_0 到 p_1 的100倍。 q_1 点可以由下面一段代

码段来获得：

```
public static final int LineScaleFactor = 100;

int dx = getP1().getX() - getP0().getX();
int dy = getP1().getY() - getP0().getY();
int x = getP0().getX() + LineScaleFactor * dx;
int y = getP0().getY() + LineScaleFactor * dy;
PointGeometry q1 = new PointGeometry(x, y);
```

类似的，点 q_0 正好位于点 p_0 的另一个方向上较远处。实现了shape方法后，试着运行一下PaintClassifyPoints程序（练习5.7）。

5.3 特化继承

特化继承是一个类继承它的父类的目的是重新定义父类中的一些方法，而其他方法保持不变。被子类重新定义的继承的方法称为被子类覆盖了。通过覆盖某些方法，子类把它父类的行为特殊化。

一个子类覆盖它的父类的方法有多个原因，下面是最常见的一些原因：

- 为了创建行为比它的父类更特殊化的新类。新类代表了一个比它的父类更特殊化的概念。
- 为了创建父类的变体。子类重用父类的元素来简化它自己的实现。
- 为了创建比它的父类更加有效的新类。新的子类展示了和它的父类相同的行为，但是比父类更加有效率，特别是使用更少的时间或空间。
- 为了创建比它的父类更强大的新类。就是说新的子类用更少的资源完成更多的工作。

用下面一种或两种方法完成：

- 1) 子类的某些方法覆盖那些前置条件较强的方法，这意味着子类实现的功能和父类一样，但是它对客户的要求更弱。
- 2) 子类的某些方法覆盖那些后置条件较弱的方法，这意味着子类比它的父类完成的功能更多，但是对客户要求没有增加。

常用特化继承的情况是：一个类描述了对象一般的行为，但有一些对象的行为需要特殊描述，所以需要定义子类覆盖其中的一些行为。其中一个例子就是MyGraphicsProgram模板，它的父类（ApplicationPanel）的makeContent方法是一个什么都不做的空方法体，这表示一般正确的行为。当你定义ApplicationPanel类的子类时，覆盖了makeContent方法，把ApplicationPanel类特化了。通常情况下，你的类还要覆盖paintComponent方法来绘制图形。没有覆盖这两个方法的子类，产生的是一个空白的框架，没有内容输出；覆盖这两个方法后，可以实现各种各样的图形输出行为（参见图5-4）。

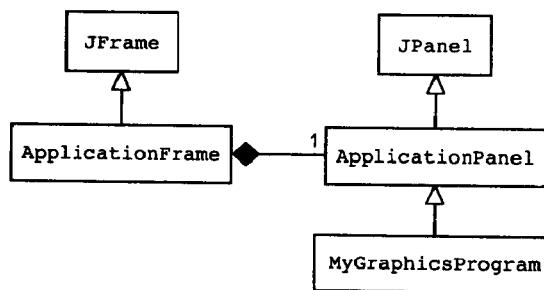


图5-4 MyGraphicsProgram特化了ApplicationPanel

5.3.1 多边形

继承的用途之一是定义一个表示和已有类相似的但不等同的抽象的类。新类是已有类的变种。使用继承实现类比从一无所有做起更容易。在这一节，我们将使用继承定义一个表示多边形的类 `PolygonGeometry`，继承折线类 `PolylineGeometry`。

多边形 (polygon) 是平面上由直线段组成的轨迹，它的第一个顶点和它的最后一个顶点是相同的。多边形和折线相似，不同的是多边形是封闭的，而折线是开放的。在一个多边形中，每一个顶点都连着多边形的两条边，所以一个有 n 个顶点的多边形有 n 条边。在一个顶点的多边形中，仅有的一条边（叫做回路 (loop)）连接顶点自己。在两个顶点的多边形中，这两个顶点被两条不同的边连接。有时候简单地用 n -gon 来表示 n 个顶点的多边形。

图 5-5 显示了多边形标识方法：一个多边形的 n 个顶点从 0 到 $n-1$ 排序，把它们标为 v_0 到 v_{n-1} ； n 条边从 0 到 $n-1$ 排序，边 e_i 连接顶点 v_i 和 v_{i+1} ($0 \leq i < n-1$)，边 e_{n-1} 连接顶点 v_{n-1} 和顶点 v_0 。虽然一个顶点的多边形 ($1-gon$) 包含一条 0 长度的边，也可以用回路表示顶点和它自己连接。图 5-5 也表示了两个顶点的多边形 ($2-gon$) 的画法，它的两条边画成了不同的曲线，在视觉上来区别它们；实际上这两条边是重合的。

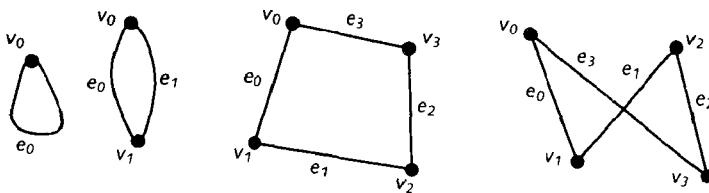


图 5-5 四种多边形

图 5-5 也说明了一个多边形的边可以交叉。没有交叉边的多边形称为简单多边形 (simple polygon)。一个带有多于两个不同顶点的多边形把平面分成两个连续区域：有限的内部区域和无限的外部区域。本书中，如果我们不特别强调就是指简单多边形。

像矩形和椭圆一样，多边形包围的区域就是多边形的内部区域 (interior)。当我们填充一个多边形或者判断一个给定点是否在多边形内部时，都是指操作多边形的内部区域。简单多边形的内部区域的定义是很直观的，就是多边形包围的连续区域。而一个交叉多边形的内部区域的定义不很明显。为了方便阐述，我们使用缠绕规则 (winding rule)。缠绕规则表明怎样判断一个点 p 是否在多边形的内部区域。Java 提供了两种不同的缠绕规则：偶 - 奇缠绕规则 (even-odd winding rule) 和非零缠绕规则 (nonzero winding rule)。

在偶 - 奇缠绕规则下，我们可以想象画一条从点 p 开始的射线。如果这条射线穿过多边形奇数次，点 p 在多边形内部；如果不是，点 p 就在多边形的外部。这个规则又称奇偶缠绕规则 (parity winding rule)，以强调穿过多边形的奇偶次数 (奇数或偶数)。强调一下，奇偶数取决于点 p 和多边形，和射线的方向无关。

为了理解非零缠绕规则，可以想象从点 p 开始画一条射线，然后从多边形的一个顶点开始，沿多边形的边一直走，直到又回到了开始的顶点，数一下从左到右穿过射线的次数，和从右到左穿过射线的次数。仅当这两个数不相等时，点 p 才位于多边形的内部。

在简单多边形中，使用两种缠绕规则定义的多边形内部区域是一致的。对于非简单多

边形，对同一个多边形使用这两种规则会产生不同的内部区域。当填充这样一个多边形时，结果图形就与缠绕规则相关。图5-6显示了同一个非简单多边形在不同的缠绕规则下填充的结果。

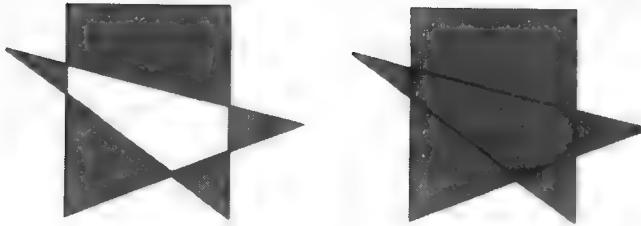


图5-6 使用偶-奇缠绕规则（左）和非零缠绕规则（右）填充的同一个多边形

除了构造器不同外，`PolygonGeometry`类的`contains`方法和`PolylineGeometry`类的定义会不同，这个方法判断给定点是否在多边形内部。另外，假设所有的多边形都使用非零缠绕规则（尽管练习5.12修改了`PolygonGeometry`的接口以便客户可以选择缠绕规则）。下面是`PolygonGeometry`的框架描述：

```
public class PolygonGeometry extends PolylineGeometry {

    public PolygonGeometry(PointGeometry[] vs)
        throws NullPointerException, ZeroArraySizeException
    // EFFECTS: If vs is null or vs[i] is null for some
    // legal i throws NullPointerException; else if
    // vs.length==0 throws ZeroArraySizeException;
    // else constructs this polygon to have the
    // vertex sequence given by vs.

    public PolygonGeometry(PolygonGeometry poly)
        throws NullPointerException
    // EFFECTS: If poly is null throws
    // NullPointerException; else initializes this
    // with the same vertex sequence as poly.

    public PointGeometry getVertex(int i)
        throws IndexOutOfBoundsException
    // EFFECTS: If 0 <= i < nbrVertices() returns
    // the position of the vertex at index i;
    // else throws IndexOutOfBoundsException.

    public void setVertex(int i, PointGeometry v)
        throws NullPointerException, IndexOutOfBoundsException
    // MODIFIES: this
    // EFFECTS: If v is null throws NullPointerException;
    // else if 0 <= i < nbrVertices() moves the vertex
    // at index i to position v in the plane;
    // else throws IndexOutOfBoundsException.

    public LineSegmentGeometry edge(int i)
        throws IndexOutOfBoundsException
    // EFFECTS: If 0 <= i < nbrEdges() returns the edge i;
    // else throws IndexOutOfBoundsException.

    public int nbrVertices()
    // EFFECTS: Returns the number of vertices
}
```

```

    // in this polygon.

public int nbrEdges()
    // EFFECTS: Returns the number of edges
    // in this polygon.

public java.awt.Shape shape()
    // EFFECTS: Returns the shape of this polygon.

public void translate(int dx, int dy)
    // MODIFIES: this
    // EFFECTS: Translates this polygon by dx along x
    // and by dy along y.

public boolean contains(int x, int y)
    // EFFECTS: Returns true if the point (x,y) is
    // contained in this polygon; else returns false.

public boolean contains(PointGeometry p)
    throws NullPointerException
    // EFFECTS: If p is null throws NullPointerException;
    // else returns true if p is contained in this
    // polygon; else returns false.

public String toString()
    // EFFECTS: Returns "Polygon: v0,v1,...,vn-1"
    // where each vi describes vertex i.
}

```

由于我们把多边形看作是在折线的基础上增加一条连接顶点 v_{n-1} 和 v_0 的边 e_{n-1} ，所以PolygonGeometry类继承PolylineGeometry类。

与折线相同，多边形可以用一个包含一个或多个点的数组来构造，它们表示多边形的顶点。一个多边形也可以用另一个多边形来构造。下面是构造器的定义：

```

public PolygonGeometry(PointGeometry[] vs)
    throws NullPointerException, ZeroArraySizeException {
    super(vs);
}

public PolygonGeometry(PolygonGeometry poly)
    throws NullPointerException {
    super(poly);
}

```

由于多边形比折线多一条边 e_{n-1} ，所以须覆盖那些必须考虑多边形中这一额外的边的方法。其中之一是edge方法。当调用edge且参数为 $n-1$ 时，需构造额外的边 e_{n-1} ；当参数小于 $n-1$ 时，调用父类的edge方法产生所需的边：

```

// method of PolygonGeometry class
public LineSegmentGeometry edge(int i)
    throws IndexOutOfBoundsException {
    if (i == nbrVertices() - 1)
        return new LineSegmentGeometry(getVertex(i),
                                       getVertex(0));
    else
        return super.edge(i);
}

```

要考虑多边形中额外的边还需要覆盖的方法是nbrEdges：

```
// method of PolygonGeometry class
public int nbrEdges() {
    return nbrVertices();
}
```

一个多边形的字符串描述符由“polygon”后面跟着按下标排列的多边形的顶点序列组成。**PolygonGeometry**的父类定义了一个保护型**VerticeToString**方法，将顶点序列转换成字符串。因为保护型方法可以被子类访问，所以可以在**toString**方法的**PolygonGeometry**版本中调用这个方法：

```
// method of PolygonGeometry class
public String toString() {
    return "Polygon: " + verticesToString();
}
```

shape方法的实现和父类的**shape**方法的实现类似。在多边形中，调用**closePath**把最后一个顶点和第一个顶点用一条边连接起来：

```
// method of PolygonGeometry class
public Shape shape() {
    GeneralPath path = new GeneralPath();
    PointGeometry v = getVertex(0);
    path.moveTo(v.getX(), v.getY());
    for (int i = 1; i < nbrVertices(); i++) {
        v = getVertex(i);
        path.lineTo(v.getX(), v.getY());
    }
    path.closePath();
    return path;
}
```

上面**shape**方法的定义在1-gon和2-gon时，没有产生图5-5那样的结果。1-gon显示为一个小点，2-gon显示为连接两个顶点的一条直线段。在5.6.4节中，我们将编写另一个类，把1-gon画成回路，2-gon画成两条不同的曲线。

在非零缠绕规则下，当一个点 $p=(x, y)$ 位于多边形的内部或者多边形的边上时，这个点才包含在一个多边形中。这意味着多边形包含它的顶点，因为它们都在多边形的边上。把判断点是否包含在多边形中的任务交给**Shape**接口**intersects**方法来完成。要实现**contains**方法，将使用**Shape.intersects**方法。**intersects**用来判断两个区域是否相交，所以这里把点

作为

一个非常小的正方形，如果这个正方形和多边形的内部或边界相交，就表明点

是在

这个多边形中：

```
// methods of PolygonGeometry class
public boolean contains(int x, int y) {
    return shape().intersects(x-0.01, y-0.01, 0.02, 0.02);
}

public boolean contains(PointGeometry p) {
    return contains(p.getX(), p.getY());
}
```

恰巧Java的**Shape**接口中定义了**contains**方法，但是它不符合我们的要求，**Shape**的点包含的概念和这里要求的不同。**Shape.contains(p)**为真，仅当点p位于内部或部分边界上。举例来说，一个与坐标轴平行的正方形仅包含了那些位于它的内部、顶边和左边的点，但是没有包含它的底边和右边。因为我们这里的包含的概念与Java的**Shape**类

不同，所以没有用Shape的contains方法，而用intersects方法来判断是否包含点。

练习

5.10 一个单步计数器是一个步值为1的N步计数器。通过扩展5.2.1节中NstepCounter类来定义OneStepCounter类。

5.11 编写一个应用程序来画一个多边形，填充用红色，轮廓是蓝色。这个应用程序的参数包含偶数个整数（不能少于4个）：

```
> java PaintPolygon x0 y0 x1 y1 ...
```

顶点 v_i 的x和y坐标在参数 $2i$ 和 $2i+1$ 的位置上。

5.12 修改PolygonGeometry类的定义，使它包括一个缠绕规则特性，它的值决定使用哪一种规则。默认的是非零缠绕规则。修改的PolygonGeometry可以定义一个域来存储这个特性的值：

```
// field of PolygonGeometry class:  
// winding rule property  
protected int windingRule;
```

以及该属性的设置者和获取者方法。WindingRule域可以为下面定义之一：

```
// static fields of the PolygonGeometry class  
static public int  
WIND_EVEN_ODD =  
    java.awt.geom.GeneralPath.WIND_EVEN_ODD,  
WIND_NON_ZERO =  
    java.awt.geom.GeneralPath.WIND_NON_ZERO;
```

为了设置缠绕规则，在返回形状以前添加语句：

```
path.setWindingRule(windingRule);
```

到shape方法中。

5.13 编写一个像练习5.11那样的图形程序，它的第一个参数决定使用哪种缠绕规则：

```
> java PaintWindPolygon winding-rule x0 y0 x1 y1 ...
```

winding-rule将以“even-odd”或“nonzero”字符串来替换。这个练习和上一个练习有关。

5.14 回忆练习4.21中维护名字-值对的Dictionary类，它的名字-值对是无序的。定义一个类OrderedDictionary来使它的字典按名字的字母排序：索引值增加时，名字字母按增序排列。这样按索引值访问的name和value方法的后置条件就加强了：

```
// methods of OrderedDictionary  
public String name(int i)  
    throws IndexOutOfBoundsException  
// EFFECTS: If 0 <= i < size() returns the name  
// of the pair at index i where the pairs are  
// ordered lexicographically by name;  
// else throws IndexOutOfBoundsException.  
  
public String value(int i)  
    throws IndexOutOfBoundsException  
// EFFECTS: If 0 <= i < size() returns the value  
// of the pair at index i where the pairs are
```

```
// ordered lexicographically by name;
// else throws IndexOutOfBoundsException.
```

[提示：你的类应该扩展Dictionary类，并覆盖其中一些方法，保证名字-值对按名字排序。可以用ComparableAttribute对象来表示名字-值对，并把它存储在集合对象中（参照练习5.6）。注意：当你比较两个可比的属性时，一定要强制它们被转换成ComparableAttribute类型。]

用OrderedDictionary代替Dictionary对象，用你在练习4.22编写的TryDictionary程序的来测试OrderedDictionary。无论何时用户输入print命令，字典对就会按序打印出来。

在这个练习中，使用继承定义了一个比它的父类更强大的类：有序字典按字母名字排序名字-值对，而原来的字典是无序排列的。]

5.3.2 标记计数器

子类通常是扩展继承和特化继承一起应用的：在5.1节开始的那个简单的Waiter和Employee类的例子就是这样；在练习5.12中实现的PolygonGeometry类也是这样的。下面再看一个关于计数器的简单例子。

标记计数器（tally counter）是一个N步计数器，它还记下了调用inc和dec方法的次数。incTally方法报告了从计数器创建开始inc方法被调用的次数，decTally方法报告了dec方法被调用的次数。

为了举例说明标记计数器的使用，可以考虑修改程序TryNStepCounter（练习5.2中）为TryTallyCounter。这个程序包含一个标记计数器，它的步由第一个参数确定；剩下的参数决定了计数器是怎样工作的：它增加 n_1 次，然后减少 n_2 次，然后增加 n_3 次，等等。每一个增加和减少设置后，计数器的值和调用inc、dec的累计和被打印出来。

下面是运行结果的一个例子：

```
> java TryTallyCounter 2 6 3 9 1
step = 2
6 incs, 0 decs: value = 12
6 incs, 3 decs: value = 6
15 incs, 3 decs: value = 24
15 incs, 4 decs: value = 22
```

每一行（除了第一行）使用下面三条语句产生：

```
System.out.print(cnt.incTally() + " incs, ");
System.out.print(cnt.decTally() + "decs: ");
System.out.println("value = " + cnt.value());
```

cnt引用TallyCounter对象。下面是TallyCounter类定义：

```
public class TallyCounter extends NStepCounter {

    protected int incTally, decTally;

    public TallyCounter(int step) {
        super(step);
        incTally = decTally = 0;
    }

    public TallyCounter() {
        this(1);
    }
}
```

```

public void inc() {
    super.inc();
    incTally++;
}

public void dec() {
    super.dec();
    decTally++;
}

public int incTally() { return incTally; }

public int decTally() { return decTally; }
}

```

TallyCounter类同时使用了扩展继承和特化继承。它覆盖了inc方法，所以它的inc方法不仅增加计数器值并且刷新incTally值。它的inc方法调用了父类的inc实现增加计数器值：

```
super.inc();
```

super关键字把当前对象看成是父类的对象。这样调用父类的inc方法完成增加计数器值的行为；然后再增加incTally：

```
incTally++;
```

同样TallyCounter类覆盖了dec方法。

TallyCounter增加了两个新方法：incTally和decTally。通过特化继承（inc和dec）、扩展继承（增加incTally和decTally方法）这两种方法共同完成标记计数器的构建。

练习

5.15 实现本节开始介绍的TryTallyCounter程序。

5.16 定义ClearableTallyCounter类来描述一个标记计数器，它可以清零（使用一个clear方法）。incTally方法报告inc方法最近被调用过的次数（计数器构造后开始计数，或最后一次调用clear开始计数）；decTally方法的工作与此类似。你的类可以扩展TallyCounter类或ClearableCounter类，这两种方法都可行吗？

5.4 说明继承

说明继承是指子类实现了它的父类或实现它的接口声明的抽象方法。通过这种实现，这个类给抽象方法提供了具体的行为。实际上，抽象方法的目的就是要子类实现它的行为。

5.4.1 接口和抽象类

Java提供了两种机制来支持说明继承：

- 一个类A可以实现一个接口或多个接口。每一个接口声明了一定数目的抽象方法而没有实现它们。当类A实现一个接口声明的方法时，类A实现了该方法指定的行为。

• 一个类A可以扩展一个抽象类。这个抽象类声明了一定数目的方法，其中一些实现了，而另外一些没有实现。这个类中没有实现的方法就是它的抽象方法（abstract method）。当类A实现了它的抽象父类声明的抽象方法时，它就实现了该抽象方法指定的行为。

当一个类实现一个接口时，它就要实现这个接口声明的方法。下面这一简短例子的类图如图5-7所示：

```
public interface I {
    public void f(int i);
    public void g(String s);
}

public class D implements I {
    public void f(int i) {
        System.out.println(i);
    }
    public void g(String s) {
        System.out.println(s);
    }
}
```

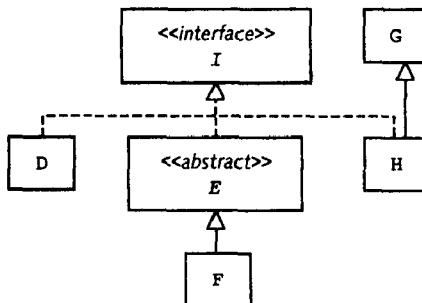


图5-7 说明继承类图

这里类D实现了接口I。接口I声明了方法f和g，但没有实现它们。它们的实现要交给实现接口的那些类（在这个例子中为类D）。因为类D实现了接口I声明的每一个方法，类D是一个具体类（concrete class），这意味着可以创建类D的实例。

并不要求一个类实现它的接口中声明的每一个方法。这里类E实现了接口I其中的一个方法f，但并没有实现另一个方法g：

```
abstract public class E implements I {
    public void f(int i) {
        System.out.println("arg = " + i);
    }
}
```

因为类E没有实现接口I声明的所有方法，类E是一个抽象类（abstract class）。类E是一个抽象类，所以它不能实例化。如果试图这样做，编译时会出错：

```
E e = new E(); // error: E is an abstract class
```

说明继承也发生在子类扩展一个抽象类的时候。抽象类声明了一个接口，但仅提供了部分方法的实现。就像接口被设计成需要子类实现一样，抽象类被设计成需要子类扩展。子类继承抽象类的说明和部分实现，且可以自由实现其中的抽象部分。类F扩展了抽象类E，并且实现了类E唯一的抽象方法：

```

public class F extends E {
    public void g(String s) {
        System.out.println(s);
    }
}

```

类F仅继承了方法g的说明，并提供了自己的实现。它同时也继承了方法f的完整实现。类F包括完整的实现，所以它是一个具体类。

一个类只可以扩展一个抽象类，但可以实现任何数目的接口。假设类G已经定义，我们可以定义一个新类H同时扩展类G和实现接口I：

```

public class H extends G implements I {
    public void f(int i) {
        System.out.println("arg: " + i);
    }
    public void g(String s) {
        System.out.println("the string is " + s);
    }
}

```

我们称Java支持多接口继承 (multiple interface inheritance)，是因为在Java中一个类可以实现任何数目的接口。一个类从它的扩展类继承了一个接口和部分实现，从它实现的每一个接口中继承一个接口。上面的例子中，类H从它的父类G继承了一个接口和部分实现，而从接口I继承了一个接口。

在图5-7图中，类型标识<<interface>>和<<abstract>>是统一建模语言 (UML) 中构造型 (stereotype) 的例子。注意用虚线连接类D、E、H到它们的接口I，这种表示法会使大家想起来I不是它们的父类，而是它们实现的一个接口。

一个类的父型包括类本身、它扩展的类和它实现的接口和它的父类的父型和接口的父型。一个类是它的每一个父型的子型。在图5-7中，类F的父型是F，E和I，类H的父型是H，I和G。接口I的子型包括图中除了类G的每一个类和接口。

子型和父型的概念是很重要的，因为一个类继承了它的每一个父型的接口。我们将在本章后面介绍多态性时探讨它的意义。

5.4.2 矩形几何图形

抽象类的设计通常是通过提取一定数目的相似类的共同行为完成的。这些共同行为被从这些类中提取出来，组合在一个新的抽象类中作为它们公共的父类，这个过程称为类分解 (factorization)。在这一节中，我们将使用类分解来定义一个新的抽象类，作为 `RectangleGeometry` 和 `EllipseGeometry` 类的公共父类。很容易看出这两个类有很多相同的行为：它们都有位置、宽度、高度特性，都定义了 `contains` 方法来决定一个给定点是否在区域内部，都有 `translate` 方法来移动图形，两个类都定义了 `shape` 方法。

因为矩形和椭圆有很多重要的区别，它们不能结合在一个类中。矩形和椭圆有不同的形状、不同的决定点包含的方法和不同的字符串描述符。

为了给两个类构造一个共同的抽象父类，我们找出两者共同的行为，并把它们放在一个新的父类中，叫做 `RectangularGeometry` 类。这个类描述了矩形几何图形 (rectangular geometry) 的概念，它描述了一个被与坐标轴平行的矩形围绕的几何图形。在 `RectangularGeometry` 中定义了那些子类共享的方法，并且把尽可能多的实现放在抽象父类中。

我们希望RectangularGeometry类实现哪些行为呢？哪些行为又只给出说明不实现呢？矩形和椭圆都有位置，宽度和高度特性，对这些特性进行访问和设置的行为是标准的，它们可以在RectangularGeometry类中实现。translate方法也可以实现，因为通过改变被所有的矩形几何图形共享的位置特性来完成。

相反，RectangularGeometry类声明了两个抽象方法：

```
// abstract methods of RectangularGeometry class
abstract public boolean contains(int x, int y);
abstract public Shape shape();
```

这些抽象方法的实现推送到子类。因为不同矩形几何图形可有不同的形状，所以RectangularGeometry类不能实现shape方法。类似的，一个矩形几何图形不能仅靠它的维数来判断它是否包含某一点。例如，假设一个矩形和一个椭圆有相同的定界框，有一些点（例如矩形的角）包含在矩形中但不在椭圆中。

translate方法可以利用位置特性实现：

```
// method of RectangularGeometry class
public void translate(int dx, int dy) {
    PointGeometry pos = getPosition();
    int x = pos.getX();
    int y = pos.getY();
    setPosition(new PointGeometry(x + dx, y + dy));
}
```

虽然RectangularGeometry类推迟了两个参数的contains方法的实现，但一个参数的contains可以在这里实现：

```
// method of RectangularGeometry class
public boolean contains(PointGeometry p) {
    return contains(p.getX(), p.getY());
}
```

被调用的两个参数的contains是一个抽象方法，它的实现由子类提供。

我们来完成RectangularGeometry类的定义：这里用两个Range对象描述一个矩形几何图形。这两个范围共同定义几何图形的定界框——矩形。这和3.2.2节中RectangleGeometry类的存储结构一样。毫不奇怪，RectangularGeometry类的实现和RectangleGeometry类的实现非常相似。下面是类定义：

```
public abstract class RectangularGeometry {

    protected Range xRange, yRange;

    public abstract boolean contains(int x, int y);
    public abstract Shape shape();

    protected RectangularGeometry(int x, int y,
                                  int width, int height)
        throws IllegalArgumentException {
        if ((width < 0) || (height < 0))
            throw new IllegalArgumentException();
        xRange = new Range(x, x + width);
        yRange = new Range(y, y + height);
    }

    protected RectangularGeometry(PointGeometry pos,
```

```

        int width, int height)
throws NullPointerException,IllegalArgumentException {
    this(pos.getX(), pos.getY(), width, height);
}

protected RectangularGeometry(Range xRange,Range yRange)
throws NullPointerException {
    this(xRange.getMin(), yRange.getMin(),
        xRange.length(), yRange.length());
}

protected RectangularGeometry(RectangularGeometry r)
throws NullPointerException {
    this (r.getPosition(), r.getWidth(), r.getHeight());
}

public PointGeometry getPosition() {
    return new PointGeometry(xRange.getMin(),
        yRange.getMin());
}

public void setPosition(PointGeometry p)
throws NullPointerException {
    xRange.setMinMax(p.getX(), p.getX() + getWidth());
    yRange.setMinMax(p.getY(), p.getY() + getHeight());
}

public int getWidth() {
    return xRange.length();
}

public void setWidth(int newWidth)
throws IllegalArgumentException {
    if (newWidth < 0) throw new IllegalArgumentException();
    xRange.setMax(xRange.getMin() + newWidth);
}

public int getHeight() {
    return yRange.length();
}

public void setHeight(int newHeight)
throws IllegalArgumentException {
    if (newHeight < 0)
        throw new IllegalArgumentException();
    yRange.setMax(yRange.getMin() + newHeight);
}

public RectangleGeometry boundingBox() {
    return new RectangleGeometry(getPosition(),
        getWidth(), getHeight());
}

public boolean contains(PointGeometry p)
throws NullPointerException {
    return contains(p.getX(), p.getY());
}

public void translate(int dx, int dy) {
    PointGeometry pos = getPosition();
    int x = pos.getX();
    int y = pos.getY();
    setPosition(new PointGeometry(x + dx, y + dy));
}

```

```

    }

    protected String dimensionsToString() {
        return getPosition() + "," + getWidth() +
            "," + getHeight();
    }
}

```

RectangularGeometry类提供了一个保护型dimensionsToString方法，让它的子类在它们自己的toString方法中使用，这个方法用字符串形式描述图形的维数。字符串描述符输出矩形几何图形类型名字（例如Ellipse）后面加上图形的维数：

```
RectangularGeometry e = new EllipseGeometry(1, 2, 3, 4);
System.out.println(e); // Ellipse: (1,2),3,4
```

椭圆的toString方法的实现调用了继承的dimensionsToString方法产生图形的字符串维数。

现在我们可以修改RectangleGeometry类的实现，使它从RectangularGeometry类扩展而来：

```

public class RectangleGeometry
    extends RectangularGeometry {

    public RectangleGeometry(int x, int y,
                           int width, int height)
        throws IllegalArgumentException{
        super(x, y, width, height);
    }

    public RectangleGeometry(PointGeometry position,
                           int width, int ht)
        throws NullPointerException,IllegalArgumentException{
        super(position, width, ht);
    }

    public RectangleGeometry(Range xRange, Range yRange)
        throws NullPointerException {
        super(xRange, yRange);
    }

    public RectangleGeometry(RectangularGeometry r)
        throws NullPointerException {
        super(r);
    }

    public boolean contains(int x, int y) {
        PointGeometry pos = getPosition();
        int minX = pos.getX();
        int minY = pos.getY();
        return (minX <= x) && (x <= minX + getWidth()) &&
            (minY <= y) && (y <= minY + getHeight());
    }

    public Shape shape() {
        PointGeometry pos = getPosition();
        return new Rectangle2D.Float(pos.getX(), pos.getY(),
                                     getWidth(), getHeight());
    }

    public Range xRange() {

```

```

        int x = getPosition().getX();
        return new Range(x, x + getWidth());
    }

    public Range yRange() {
        int y = getPosition().getY();
        return new Range(y, y + getHeight());
    }

    public String toString() {
        return "Rectangle: " + dimensionsToString();
    }
}

```

RectangleGeometry类实现了由其父类声明的抽象方法shape和contains。

练习

5.17 重新实现EllipseGeometry类，使它扩展RectangularGeometry类。

5.4.3 几何图形抽象

在这一节中，将继续上一节中所开始的RectangularGeometry类的分解过程，最终目的是设计出如图5-8所示的几何图形的继承层次结构图。这个继承层次结构在本书中将保持不变。

图5-8中包含了两个新的接口：AreaGeometry和Geometry。Geometry接口声明了两个通用操作来获得图形形状（shape）和移动图形（translate），下面是它们的定义：

```

public interface Geometry {
    public abstract java.awt.Shape shape();
    // EFFECTS: Returns this geometry's shape.

    public abstract void translate(int dx, int dy);
    // MODIFIES: this
    // EFFECTS: Translates this geometry by dx
    //           along x and dy along y.
}

```

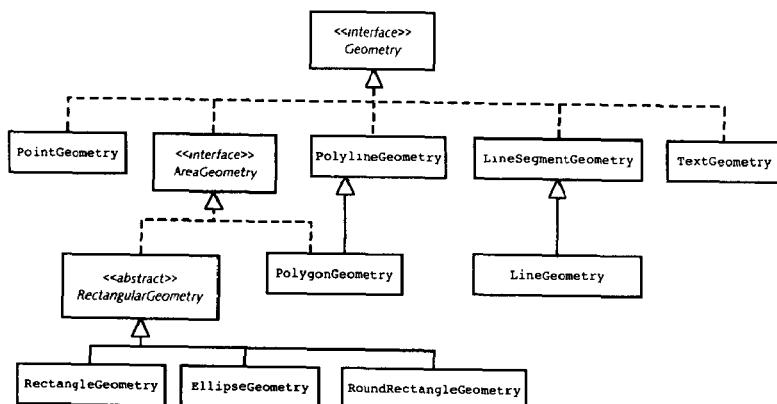


图5-8 Geometry子类的类图

AreaGeometry接口表示包含封闭区域的几何图形。到目前为止，我们遇到了三种此类图形：矩形、椭圆和多边形。它们都定义判断点是否包含在区域中的方法contains。

AreaGeometry接口定义如下：

```
public interface AreaGeometry extends Geometry {
    public abstract boolean contains(int x, int y);
    // EFFECTS: If this geometry contains (x,y) -- that
    // is, (x,y) lies in this geometry's interior or
    // boundary -- returns true; else returns false.

    public abstract boolean contains(PointGeometry p)
        throws NullPointerException;
    // EFFECTS: If p is null throws NullPointerException;
    // else if this geometry contains p returns true;
    // else returns false.
}
```

AreaGeometry接口是从**Geometry**接口扩展而来，接口扩展使用关键字**extends**。只有类才可以实现接口，接口不能实现其他接口。

一个普通的概念是用接口还是抽象类来表达，需要综合考虑，接口和抽象类各有特点。Java支持接口的多重继承。这意味着一个类可以继承任意个接口。这种语言特征给使用接口带来了很多好处。比较而言，抽象类有一个好处，那就是可以为后代提供部分实现，这使得子类实现更加简单且更不易出错。然而Java只支持类单一继承，一个子类只可以扩展一个父类，所以抽象类的子类不能继承其他类或抽象类的实现。

已经很清楚，**Geometry**作为一个接口来实现最好。由于**Geometry**接口表达的几何图形太抽象，因而不能具体地（获得图形形状**shape**和移动图形**translate**）实现任一个方法。但**AreaGeometry**不是这样，它的两个**contains**方法中的任何一个都可以由另外一个来实现，即它可以提供部分方法实现，可以考虑把它作为一个抽象类来定义。例如，可以像下面这样定义抽象类**AbstractAreaGeometry**：

```
public abstract class AbstractAreaGeometry
    implements Geometry {
    public abstract boolean contains(int x, int y);

    public boolean contains(PointGeometry p)
        throws NullPointerException {
        return contains(p.getX(), p.getY());
    }
}
```

通过这种方法，**AbstractAreaGeometry**类的后代只需实现两个参数的**contains**方法，而直接继承已经实现的一个参数的**contains**方法（**RectangularGeometry**也是用这种方式处理其**contains**方法的）。可是如果我们把**AbstractAreaGeometry**类定义成抽象类，有时候会产生麻烦。如果一个子类只想继承由**AbstractAreaGeometry**声明的接口而需要从其他类继承实现，现在这种定义法就无能为力了。图5-8中有一个类似的例子：**PolygonGeometry**类从**AreaGeometry**类继承了部分接口，而从**PolylineGeometry**类继承了实现。遵循这样一个原则：如果抽象类提供的实现的功能很弱，最好把它定义成接口。

当然最好的解决方法是同时定义抽象类和接口：如图5-9所示，同时定义了**AreaGeometry**接口和实现这一接口的抽象类**AbstractAreaGeometry**。如果子类不需要别的父类的实现，则可以扩展**AbstractAreaGeometry**类，从而可以获益于**AbstractAreaGeometry**的实现。图5-9中的**RectangularGeometry**类就是这样做的。

相反，那些需要从别的父类继承实现的子类仍然需要实现AreaGeometry接口，就像PolygonGeometry类一样。

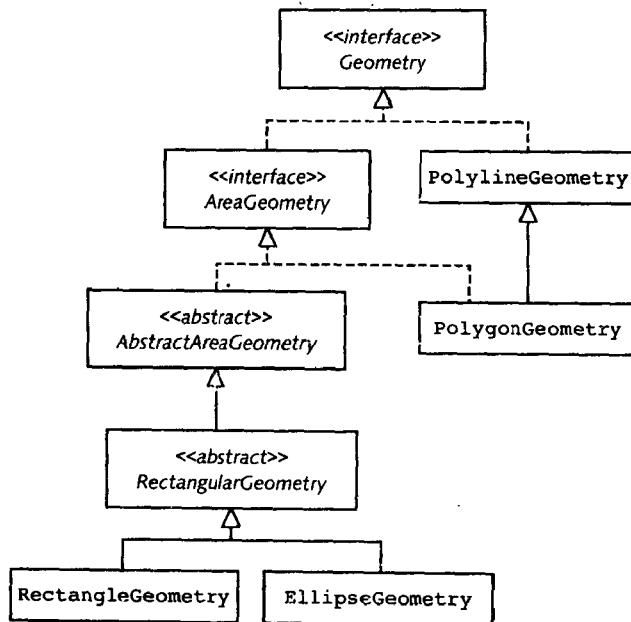


图5-9 本设计中AbstractAreaGeometry提供了AreaGeometry接口的部分实现

练习

5.18 修改几何图形类，使它能实现图5-8中所示的类继承层次结构。修改PointGeometry类、LineSegmentGeometry类和PolylineGeometry类的头定义，使它们实现Geometry接口；并且修改RectangularGeometry类和PolygonGeometry类的头定义，使它们实现AreaGeometry接口。另外，RectangularGeometry类和EllipseGeometry类中需要增加一个以RectangularGeometry对象为参数的构造器，例如：

```

public EllipseGeometry(RectangularGeometry r)
    throws NullPointerException
    // EFFECTS: If r is null throws
    // NullPointerException; else constructs an
    // ellipse with the same dimensions as r.
  
```

还需要其他的修改吗？（圆角矩形和文本图形将在下两个练习中讨论。）

5.19 圆角矩形是具有圆形拐角的矩形。利用一个椭圆使矩形拐角圆形化，这个椭圆长和宽的大小分别由arcw和arch两个特性指定（如图5-10所示）。我们这个圆角矩形提供的很多操作是从它的父类RectangularGeometry继承的。下面的这个类框架只显示了那些必须实现的方法：

```

public class RoundRectangleGeometry
    extends RectangularGeometry {

    public RoundRectangleGeometry(int x, int y,
        int width, int height,
  
```

```

        double arcw, double arch)
throws IllegalArgumentException
// EFFECTS: If width, height, arcw, or arch is
// negative throws IllegalArgumentException;
// else constructs a round rectangle at
// position (x,y) and with given width and
// height, and with corner-rounding ellipse
// of width arcw and height arch.

public RoundRectangleGeometry(Position pos,
                               int width, int height,
                               double arcw, double arch)
throws NullPointerException,
      IllegalArgumentException
// EFFECTS: If pos is null throws
// NullPointerException; else if width,
// height, arcw, or arch are negative throws
// IllegalArgumentException; else constructs
// a round rectangle at position (x,y) and
// with given width and height, and with
// corner-rounding ellipse of width arcw
// and height arch.

public RoundRectangleGeometry(Range xRange,
                             Range yRange, double arcw, double arch)
throws NullPointerException,
      IllegalArgumentException
// EFFECTS: If xRange or yRange is null
// throws NullPointerException; else if arcw
// or arch is negative throws
// IllegalArgumentException; else constructs
// a round rectangle of given x
// and y extents, and with corner-rounding
// ellipses of width arcw and height arch.

public RoundRectangleGeometry(
    RectangularGeometry r,
    double arcw, double arch)
throws NullPointerException,
      IllegalArgumentException
// EFFECTS: If r is null throws
// NullPointerException; else if arcw or arch
// is <0 throws IllegalArgumentException;
// else constructs a round rectangle with
// the same dimensions as r, and with corner-
// rounding ellipses of width arcw and
// height arch.

public boolean contains(int x, int y)
// EFFECTS: If this geometry contains (x,y)
// returns true; else returns false.

public Shape shape()
// EFFECTS: Returns this geometry's shape.

public double getArcw()
// EFFECTS: Returns the width of the
// corner-rounding ellipse.

public void setArcw(double newArcw)

```

```

throws IllegalArgumentException
// MODIFIES: this
// EFFECTS: If newArcw < 0 throws
//   IllegalArgumentException; else sets the
//   width of the corner-rounding ellipses
//   to newArcw.

public double getArch()
// EFFECTS: Returns the height of the
//   corner-rounding ellipse.

public void setArch(double newArch)
throws IllegalArgumentException
// MODIFIES: this
// EFFECTS: If newArch < 0 throws
//   IllegalArgumentException; else sets the
//   height of the corner-rounding ellipses
//   to newArch.

public String toString()
// EFFECTS: Returns the string
//   "Round-rectangle: (x,y),width,height,
//   arcw,arch".
}

```

试着实现RoundRectangleGeometry类。可以利用java.awt.geom.RoundRectangle2D.Double类来构造圆角矩形的形状：

```

// method of RoundRectangleGeometry class
public Shape shape() {
    PointGeometry pos = getPosition();
    return new RoundRectangle2D.Double(pos.getX(),
        pos.getY(), getWidth(), getHeight(),
        getArcw(), getArch());
}

```

如果想实现具有两个参数的contains方法，你可以委托给圆角矩形的shape方法完成。

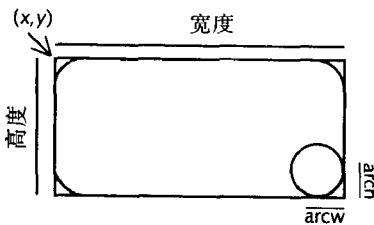


图5-10 圆角矩形

5.20 在Java中按图形输出字符串是十分简单的：发送drawString消息给绘图环境，并且把要输出的字符串和字符串开始位置的x和y坐标作为参数传入。例如，下面的语句就把字符串“hello world”输出到绘图环境g2中：

```
g2.drawString("hello world", 50, 100);
```

第一个字符“h”的基线出现在用户空间的(50, 100)位置。所画文本的其他特征值，比如字体和画笔，是由g2的当前属性指定的。

下面的例子描述了一个字符串类，字符串实现Geometry接口。下面的代码段

把字符串“hello world”输出到绘图环境中，起始位置是(50, 100)：

```
Geometry s = new TextGeometry("hello world");
s.translate(50, 100);
g2.fill(s.shape());
```

实现TextGeometry类为何不能直接用Graphics2D.drawString方法呢？drawString级别太高，要求传给它字符串，而Geometry接口类只能提供shape方法，提供图形的形状，而没有办法取出图形对应的字符串。我们的TextGeometry类的实现遵循drawString方法用来完成这个工作。

画字符串的时候，字符串首先被转换成一个称为字形轮廓(glyph)的形状矢量。这些字形轮廓描述字符串可视化外观，它不仅按字符序列排序，而且要保存所选字体、字体大小、字体格式和指定字符间隔等要素。发送“CreateGlyphVector”消息给Font对象就可以创建字形轮廓矢量。Font对象描述字体类型、字体格式(包括无格式、加粗、斜体或粗斜体)、字体大小和布局。创建字形轮廓矢量时，Font对象需要一个字体绘图环境(font rendering context)，它存储的信息包括文本大小和绘图提示等。除了注意这里所描述的内容是由下面类定义封装的之外，关于字符串按图形输出的内容不再详细介绍：

```
public class TextGeometry implements Geometry {

    // default font if not supplied to constructor
    public static final Font DefaultFont =
        new Font("SansSerif", Font.BOLD, 12);

    // glyphs for this string
    protected GlyphVector glyphs;

    // position of text baseline of first character
    protected int x, y;

    public TextGeometry(Font font, String s)
        throws NullPointerException {
        // EFFECTS: If font or s is null throws
        // NullPointerException; else constructs
        // this for string s in the given font.
        FontRenderContext frc =
            new FontRenderContext(null, true, false);
        glyphs = font.createGlyphVector(frc, s);
        x = y = 0;
    }

    public TextGeometry(Graphics2D g, String s)
        throws NullPointerException {
        // EFFECTS: If g or s are null throws
        // NullPointerException; else constructs this
        // for s using the font and font rendering
        // context in the graphics context g.
        FontRenderContext frc = g.getFontRenderContext();
        Font font = g.getFont();
        glyphs = font.createGlyphVector(frc, s);
        x = y = 0;
    }

    public TextGeometry(String s)
        throws NullPointerException {
```

```

    // EFFECTS: If s is null throws
    //   NullPointerException; else constructs
    //   this for string s using the default font.
    this(DefaultFont, s);
}

public Shape shape() {
    // EFFECTS: Returns the shape of this text.
    return glyphs.getOutline(x, y);
}

public void translate(int dx, int dy) {
    // MODIFIES: this
    // EFFECTS: Translates this text by dx and dy.
    this.x += dx;
    this.y += dy;
}
}

```

类GlyphVector和FontRenderingContext包含在java.awt.font包中。在这个练习中，写一个图形程序，目的是练习使用类RoundRectangleGeometry和TextGeometry。程序要填充一个红色圆角矩形，而且在它的下方输出一条“See my curves!”消息。圆角矩形的位置和尺寸由程序参数指定。如果s是TextGeometry对象，看一下填充s

g2.fill(s.shape())

和绘制s：

g2.draw(s.shape())

之间有何不同。

5.5 多态性

多态性 (polymorphism) 这个词出自希腊文，原始含义是许多形态。在对象模式中，多态性描述了这样一种情况：软件元素可以采用多种形态中的任何一种。特别地，通过多态性 (也称子型多态性 (subtype polymorphism)) 一个类型可以被看成是它的任何子型。使用一个类型的接口的客户可以使用它的任何子型的实例。

5.5.1 Java的多态性机制

为了解释多态性，将使用5.1节开始时介绍的Employee类和Waiter类。在这里再重复一下它们的内容：

```

public class Employee {
    public void greeting(String name) {
        // MODIFIES: System.out
        // EFFECTS: Prints a friendly greeting to name.
        System.out.println("Welcome to Eat and Run, "+name+".");
    }
    public void farewell(String name) {
        // MODIFIES: System.out
        // EFFECTS: Prints a farewell to name.
        System.out.println("Please come back soon, "+name+"!");
    }
}

```

```

class Waiter extends Employee {
    public void greeting(String name) {
        System.out.println("Can I take your order, "+name+"?");
    }
    public void recommendation() {
        // MODIFIES: System.out
        // EFFECTS: Prints a recommendation to name.
        System.out.println("I'd stick to the pizza.");
    }
}

```

在Java中使用对象时都会出现两个类型：对象的实际类型（actual type）和它的表现类型（apparent type）。对象的实际类型是指对象所属的类；对象的表现类型是指引用该对象的表达式的类型。例如，考虑如下代码段：

```

Employee e = new Waiter(); // line 1
e.greeting("David"); // line 2: Can I take your order, David?

```

在第一行中创建的Waiter对象在第二行中使用。第二行中引用Waiter对象的表达式e的类型由第一行中声明为Employee决定。在第二行中，变量e引用的对象的实际类型是Waiter且表现类型是Employee。

牢记对象创建时，对象实际类型也就创建了，而且在对象的整个生存期中都是不变的。相反对象的表现类型和表达式相关，并且可以从一个表达式转换成另一个表达式。假设我们继续先前的代码段如下：

```

Waiter w = (Waiter)e; // line 3
w.recommendation(); // line 4: I'd stick to the pizza.

```

在第四行中，表达式w引用的对象拥有相同实际类型和表现类型，都是Waiter。

编译器可以推导出对象的表现类型。在第二行中，e的表现类型是从第一行的变量e的声明中得出的。在第四行中，w的表现类型是从第三行的变量w的声明中推出的。

通常对象通过引用变量来使用，上面的代码就是如此，但使用其他方法引用对象也是可以的。例如，可以不使用对象引用，在创建对象时立即使用对象：

```
int len = new String("hello").length();
```

表达式new String ("hello")引用的对象拥有相同实际类型和表现类型String。表达式也可以进行类型转换。例如，下面的表达式：

```
(Employee)new Waiter()
```

对象除了拥有实际类型Waiter，还拥有由于类型转换而产生的表现类型Employee。

对象的真实类型限制它的表现类型：对象的表现类型总是对象真实类型的父型。前面例子中都是如此。对象的表现类型（Employee）是对象真实类型（Waiter）的父型；对象的引用类型（Waiter）是对象真实类型（Waiter）的父型。一个类型也是自身的父型。相反下面的初始化在编译中是不合法的：

```
Employee e2 = new PointGeometry(4, 5); // illegal
```

表达式e2不能被合法使用：对象的表现类型（Employee）不是对象真实类型（PointGeometry）的父型。

使用一个对象时，它的真实类型和表现类型是在两个关键规则下起作用的。第一个规则涉及到对象的接口：使用对象时，对象的表现类型定义了对象的接口。例如，由于

`greeting`方法为`Employee`类的接口的一部分，所以下面代码中第六行是合法的：

```
Employee e = new Waiter(); // line 5
e.greeting("David"); // line 6: Can I take your order, David?
e.recommendation(); // line 7: illegal
```

而`recommendation`方法不是`Employee`类的接口，所以第七行是非法的。虽然对象的真实类型把`recommendation`方法作为接口的一部分。

第二个规则涉及到对象的行为：使用对象时，对象的真实类型定义了对象的行为。再看上面例子的第六行：`e`引用的对象发送`greeting`消息，虽然`Employee`类和`Waiter`类都定义了一个`greeting`方法，但是`Waiter`类的方法获得了执行。很明显，这个行为是由对象的真实类型`Waiter`决定的，而不是表现类型`Employee`决定的。

这两个规则可总结如下：

使用对象时，对象的表现类型决定了对象的接口，对象的真实类型决定了对象的行为。

这个基本原则形成了多态性的基础。客户代码通过定义对象的公有接口的公共父型来控制对象，但是，实际执行时，每个对象按照它的真实类型响应消息。例如，考虑下面的过程：

```
static void servileGreeting(Employee e, String name) {
    System.out.println(name + ", how lovely to see you.");
    e.greeting(name);
}
```

当程序`servileGreeting`被调用的时候，它的参数`e`绑定到属于`Employee`的某个子型的一个对象。父型`Employee`实现了`greeting`接口，所以此程序可以合法地向`e`发送`greeting`消息。但是，`greeting`消息所产生的行为是由`e`引用对象的真实类型决定的。过程调用：

```
servileGreeting(new Employee(), "Elisa");
```

结果输出如下：

```
Elisa, how lovely to see you.
Welcome to Eat and Run, Elisa.
```

相比较，程序调用：

```
servileGreeting(new Waiter(), "Phyllis");
```

结果输出如下：

```
Phyllis, how lovely to see you.
Can I take your order, Phyllis?
```

这儿有一个更有用的例子。把一个绘图环境`g2`和一个几何图形数组作为参数调用`fillGeometries`过程时，它会把每个几何图形都画到绘图环境中：

```
static void fillGeometries(Graphics2D g2,
                            Geometry[] geoms) {
    for (int i = 0; i < geoms.length; i++) {
        Geometry geom = geoms[i];
        g2.fill(geom.shape());
    }
}
```

在`for`循环体中，赋值语句：

```
Geometry geom = geoms[i];
```

是合法的，这是因为存储在`geoms[i]`中的对象属于`Geometry`接口的某个子型。由于`Geometry`接口声明了方法：

```
public Shape shape();
```

所以向`geom`引用的对象发送`shape`消息是合法的：

```
geom.shape()
```

过程`fillGeometries`并不知道存储在数组中几何图形的真实类型。而且它也没有必要知道，因为`Geometry`的每个子型都返回和实际类型相符的形状：一个`EllipseGeometry`返回一个椭圆形形状；一个`PointGeometry`返回一个点形状；以此类推。每个对象的表现类型（`Geometry`）提供了`fillGeometry`要求的接口，而每个对象的真实类型提供了`fillGeometry`所期望的行为。

5.5.2 Java的Comparable接口与排序

在2.4节中，我们曾设计了一个可输入一串整型参数的程序，它把参数排序之后打印出来。程序中参数作为整型是用操作符`<`来排序。实际上，只要定义了线性排序操作（比如`<`），任何类型的变量都可以进行线性排序。为了便于定义数据类型的线性排序操作，Java提供了`java.lang.comparable`接口：

```
public interface java.lang.Comparable {
    public int compareTo(Object obj)
        throws ClassCastException;
    // EFFECTS: If this object and obj cannot be
    // compared throws ClassCastException; else
    // returns a negative integer, zero, or positive
    // integer if this object is less than, equal to,
    // or greater than obj, respectively.
}
```

`CompareTo`方法把这个对象和`obj`对象相比较，如果这个对象小于、等于或者大于`obj`对象，则分别返回负数、零和正数。实现了`Comparable`接口的任何类型都可以进行线性排序。例如，类`String`实现了`Comparable`接口，它的`compareTo`方法按字典排序法比较两个字符串：

```
"apple".compareTo("banana")      // evaluates to -1
"banana".compareTo("apple")      // evaluates to 1
"apple".compareTo("apple")       // evaluates to 0
```

类`Rational`（4.4.3节）也自己定义了一个`compareTo`方法。为了使类`Rational`类实现`Comparable`接口，可以用如下方法修改类定义的头：

```
public class Rational implements Comparable {
```

下面我们将定义一个`Sort`类，它提供一个静态`sort`方法来对`Comparable`对象数组进行排序。我们将使用2.4节中描述的选择排序方法。下面是这个类：

```
public class Sort {
    public static void sort(Comparable[] a)
        throws NullPointerException, ClassCastException {
        // MODIFIES: a
        // EFFECTS: If a is null throws NullPointerException;
```

```

// else if some pair of elements in a cannot be
// compared throws ClassCastException;
// else sorts the elements of a.
if (a == null) throw new NullPointerException();
sort(a, a.length);
}

protected static void sort(Comparable[] a, int n)
throws ClassCastException {
// REQUIRES: 0 <= n <= a.length.
// MODIFIES: a
// EFFECTS: If some pair of elements in a cannot
// be compared throws ClassCastException;
// else sorts the elements of a[0..n-1].
for (int i = 0; i < n; i++) {
    int indx = min(a, i, n-1);
    Comparable temp = a[i];
    a[i] = a[indx];
    a[indx] = temp;
}
}

protected static int min(Comparable[] a,
int lo, int hi)
throws ClassCastException {
// REQUIRES: 0 <= lo <= hi < a.length.
// EFFECTS: If some pair of elements in a cannot
// be compared throws ClassCastException; else
// returns the index of some smallest element
// in a[lo..hi].
int indx = lo;
for (int i = lo+1; i <= hi; i++)
    if (a[i].compareTo(a[indx]) < 0)
        indx = i;
return indx;
}
}

```

Sort和2.4节中的整型排序程序只有一个地方不同：Sort用compareTo方法对Comparable对象进行比较，2.4节中用操作符<对int变量进行比较。在上面代码中，方法compareTo只用在过程min中，用来在子数组中查找最小对象的位置。

方法Sort.sort不能对原始类型（如整型）进行排序。因为原始类型不能实现Comparable接口。它们不是对象，当然不能实现接口。但Java中提供包装类（wrapper classe），可以把原始类型转换成对象。包装类依据客户的需求提供适当的接口。

Java对每一个原始类型都提供了包装类。例如，Java的Integer类是用来把一个int变量转换成一个对象。表达式

```
Integer iObj = new Integer(7);
```

创建了一个新的表示7的Integer对象。类Integer实现了Comparable接口，所以可以compareTo方法实现比较大小的操作：

```
iObj.compareTo(new Integer(9)) // -1 since 7 < 9
iObj.compareTo(new Integer("-6")) // 1 since 7 > -6
```

但int数据类型提供的操作只有很少一部分被类Integer支持。例如，对象Integer不能用来加或者乘。Integer提供方法intValue来提取相等的int值：

```

int i = iObj.intValue();
System.out.println("i = " + i); // i = 7

```

下面名为SortIntegerArguments的程序和2.4节中SortIntegerArgs程序功能相同：输入整数参数序列，按递增排序输出。不同的是这个程序使用方法Sort.sort来完成排序。它先把程序输入参数转换成Integer对象，再存到一个数组中，接着调用Sort.sort对数组排序，并且从左到右打印出数组。下面是实现：

```

public class SortIntegerArguments {
    public static void main(String[] args) {
        Integer[] a = getIntegers(args);
        Sort.sort(a);
        printIntegers(a);
    }

    static Integer[] getIntegers(String[] args) {
        Integer[] a = new Integer[args.length];
        for (int i = 0; i < a.length; i++)
            a[i] = new Integer(args[i]);
        return a;
    }

    static void printIntegers(Integer[] a) {
        for (int i = 0; i < a.length; i++)
            System.out.print(a[i] + " ");
        System.out.println();
    }
}

```

类Sort拥有多态性的一般特性。它的方法使用传入对象的compareTo方法比较两个对象的大小。因为这些对象都实现了Comparable接口，所以它们都提供compareTo操作。然而在编译时并不知道被排序对象的真实类型。它们可能是Integer对象、Rational对象、String对象或者其他类型的对象。

练习

5.21 (重点) 修改类Rational定义的第一行，使它实现Comparable接口。然后编写一个程序SortRationalArguments，它有偶数个参数并且把每一对看成一个有理数：第*i*个有理数的分子和分母分别是程序的第2*i*个和第2*i*+1个参数。程序把这些有理数进行递增排序并且打印出排序结果。下面是一个运行范例：

```

> java SortRationalArguments 1 2 3 4 1 3 1 4 1 9 1 8 2 5
1/9 1/8 1/4 1/3 2/5 1/2 3/4

```

5.22 用一个Sort类来编写一个应用程序，把输入参数作为字符串来排序，并打印出来：

```

> java SortStringArguments did gyre and gimble in the wabe
and did gimble gyre in the wabe

```

5.23 (重点) 用如下方法实现平面上点的排序。设有 $p=(x_p, y_p)$, $q=(x_q, y_q)$ 。在这个排序中，当且仅符合下列条件时，点 p 小于点 q ：

- 1) $x_p < x_q$
- 2) $x_p = x_q$ 并且 $y_p < y_q$

修改类PointGeometry，以便它实现Comparable接口，它的compareTo方法必须实现按上面描述比较两个点。编写如下两个程序进行测试：

(a) 编写一个应用程序, 它有 $2n$ 个参数, 为 n 个点的坐标:

```
> java SortPoints x1 y1 x2 y2 ... xn yn
```

程序要按增序输出 n 个点坐标。例如:

```
> java SortPoints 5 2 3 1 3 7 2 9 5 8
(2,9) (3,1) (3,7) (5,2) (5,8)
> java SortPoints 3 4 hi there
Error: argument hi is badly formed.
> java SortPoints 1 2 3
Error: requires an even number of arguments.
```

(b) 编写图形程序PointSortedPoints, 它有整型参数 n ; 然后产生 n 个随机点, 并把它们排序。每个点的颜色与它在排序中的位置有关, 具体来说, 第 n 个点的颜色设置如下:

```
Color.getHSBColor(i / (float)n, 1, 1)
```

这个颜色有最大饱和度和亮度, 它的色度由浮点变量 $i / (float)n$ 决定。在框架中出现的应该是从左到右 n 个随机点。假设颜色谱范围是从红、黄、绿、蓝绿、蓝、紫红, 再回到红。

5.5.3 替代原则

使用对象时, 对象的接口由它的表现类型决定。客户代码要求对象和它的接口保持一致。这就要求遵循替代原则 (substitution principle):

在不影响客户代码正确性的前提下, 在任何要求父型出现的地方, 都可以使用子型对象。客户代码只依赖于父型提供的说明。

满足替代原则的继承层次允许客户代码只写父型的说明。可是Java编译器没有(也不可能)强制遵守替代原则, 并且很容易定义继承层次来破坏它。这个规则要程序员自己遵守才行。

回想下面例子, 来看输出问候的方法:

```
static void servileGreeting(Employee e, String name) {
    // MODIFIES: System.out
    // EFFECTS: Prints a fawning, friendly greeting to name.
    System.out.println(name + ", how lovely to see you.");
    e.greeting(name);
}
```

`servileGreeting`方法可以合法地发送`greeting`消息给`e`, 因为`greeting`方法是由`Employee`类型的接口指定的, 而它并不知道`e`引用对象的真实类型, 只知道它属于`Employee`的某一个子型。`servileGreeting`的正确性依赖于这样的期望: `Employee`的每个子型都应该按照说明定义它的`greeting`方法。`servileGreeting`期望`e`输出一个友好的问候, 用`Waiter`对象调用`ServileGreeting`时, 真的输出这样的问候:

```
servileGreeting(new Waiter(), "Karen");
// Karen, how lovely to see you.
// Can I take your order, Karen?
```

扩展一个类的时候, 子类继承父类的接口。子类覆盖的方法必须和父类的说明保持一致。这样客户代码的正确性就可以依赖于父类的这些说明。`Waiter`类覆盖`greeting`方

法时，它保持和Employee类的方法的说明一致：输出一个友好问候。Waiter满足替代原则。

有必要来看一个违背替代规则的例子。假设我们这样定义一个Cook子类：

```
class Cook extends Employee {
    public void greeting(String name) {
        System.out.println("Get the hell out of my kitchen!");
    }
}
```

当以Cook对象作为参数调用servileGreeting时，Karen得到了一个很不友好的问候

```
servileGreeting(new Cook(), "Karen");
// Karen, how lovely to see you.
// Get the hell out of my kitchen!
```

问题在于，Cook类覆盖greeting方法时，违背这个方法要求的说明：输出友好问候，而它却输出一个不友好的问候。Cook类违背了替代原则。

方法的说明是由方法的前置条件和后置条件描述的。在Waiter和cook例子中，方法greeting的说明不严格归咎于“友好”这个词含义模糊：它在堪萨斯州是一个意思，而在纽约却是另外一个意思。让我们看一个严格说明的例子。containsInIntersection过程有三个参数：点p和实现AreaGeometry接口的几何图形a和b；程序判断点p是否包含在两个图形a和b的交集中，如果是，返回true。下面是这个过程的实现：

```
static boolean containsInIntersection(PointGeometry p,
                                      AreaGeometry a, AreaGeometry b)
    throws NullPointerException {
    // EFFECTS: If p, a, or b is null throws
    // NullPointerException; else if a∩b contains p
    // returns true; else returns false.
    return a.contains(p) && b.contains(p);
}
```

containsInIntersection并不知道a和b对象的真实类型。它们也许是RectangleGeometry、EllipseGeometry、PolygonGeometry的实例或者其他任何AreaGeometry子类的实例。过程的正确性并不依赖于a或b的真实类型，它的正确性仅依赖于这些输入对象是否正确实现AreaGeometry接口描述的contains方法：

```
public abstract boolean contains(PointGeometry p)
    throws NullPointerException;
// EFFECTS: If p is null throws NullPointerException;
// else if this geometry contains p returns true;
// else returns false.
```

每种几何图形都以自己的方式实现contains方法，但是每一个实现都要和作用子句相一致。过程的正确性依赖于它调用的AreaGeometry对象的contains方法的实现是否和说明保持一致。

子型中的方法必须满足父型中的方法说明，但是这并不意味着这些说明必须是完全相同的。这意味着，每个子类方法的前置条件不应该比父型指定的强；后置条件不应该比父型指定的弱。换句话说，允许子类比父类完成得多：

- 如果子类的某个方法的前置条件比父型的要求还要弱，则方法要求客户的少，但完成的任务不会少。

- 如果子类的某个方法的后置条件比父型的要求还要强，则方法完成较多的任务，但没有要求客户更多。

通过父型使用子型对象的客户的正确性并不会受子型对象的不同的影响。既然父型的前置条件能满足客户的要求，那么比父型弱的子型的前置条件也一定能满足客户的要求；与此相似，既然父型的后置条件能满足客户的要求，那么比它强的子型的后置条件也一定能满足客户的要求。

请看类似的银行例子。假设你在一个叫SuperBank的银行开了一个账户，银行账户有如下两个条款：

1'前置条件：你同意保存最小余额\$500。

2'后置条件：银行承诺每年付5%利息。

假设在你不知道的情况下，银行把你的钱转储到第二个名叫SubBank的银行。（在这个类比中，你是客户，使用父型SuperBank，而你实际使用的对象真实类型是SubBank。）在没有通知你的情况下，SubBank更改条款如下：

1'前置条件：同意保存最小余额\$250。

2'后置条件：银行承诺每年付6%利息。

这样，SubBank削弱了客户必须满足的前置条件，并加强了它的承诺。条款中这样的更改无疑不会引起任何不便。事实上，如果你从不检查账目表，你将完全不会察觉这种更改。你将保存最小余额\$500，符合前置条件1'；并且你将在每年5%的担保基础上收回利息，这也完全符合后置条件2'。你的期望是由你和SuperBank之间的协定满足的，然而你的期望是由SubBank完成的。因此，SubBank满足了父型SuperBank的说明：它的前置条件1'不比SuperBank的前置条件1强；它的后置条件不比SuperBank的后置条件弱。

类OrderedDictionary（练习5.14）是一个比它的父类（Dictionary）强的例子。OrderedDictionary按字母对它的名字-值对进行排序，Dictionary却没有这样做，其他功能一样。当一个客户通过父型使用OrderedDictionary按索引访问名字-值对时，它获得的是有序对，虽然它期望无序对也可以。OrderedDictionary按索引对名字-值对访问的方法有比父类的方法更强的后置条件，而且它们满足了Dictionary类的说明以及客户的要求。

无论何时定义一个新类，都要使用替代原则来确保和它的父型保持一致。定义一个新类B实现父型A的某个方法f，你可以用较弱的前置条件，和/或较强的后置条件。换句话说，B.f的前置条件不能比A.f的前置条件强；B.f的后置条件不能比A.f的后置条件弱。

练习

5.24 再看银行例子，可以设想这样的情况：在没有通告你的情况下，SuperBank转约到一个SubGreedyBank。SubGreedyBank银行修改存款条款如下：

1"前置条件：同意存最小余额\$1000。

2"后置条件：银行承诺每年付4%的利息。

为什么这样就违背了替代原则？是否1"或2"单独违背了这个原则？

5.25 把这章中的计数器类的类图画出来。看一下是否每个计数器类都遵循替代原则？

5.26 考虑下边一个计数器类的说明，此计数器增时加1，减时减2：

```
public class DoubleStepCounter
```

```

    extends NStepCounter {
    public DoubleStepCounter(int incStep,
                             int decStep)
        // EFFECTS: Initializes this counter's value
        // to zero, and initializes the increment
        // step to incStep and the decrement step
        // to decStep.

    public void inc()
        // MODIFIES: this
        // EFFECTS: Increments this counter's
        // value by incStep.

    public void dec()
        // MODIFIES: this
        // EFFECTS: Decrements this counter's
        // value by decStep.

    public int step()
        // EFFECTS: Returns incStep.

    public int value()
        // EFFECTS: Returns this counter's value.
    }
}

```

DoubleStepCounter类是否满足替代原则？为什么是或不是？

5.27 替代原则有时会阻止我们使用继承，而这种情况看起来似乎使用继承好像没有问题，实际上会有问题。例如，正方形是矩形的一种，我们希望定义一个类SquareGeometry扩展RectangleGeometry类，这个类表示正方形。为了保证边长都是相等，需要覆盖setWidth和setHeight方法。这两个方法可以声明如下：

```

// methods of SquareGeometry class
public void setWidth(int newSide)
    throws IllegalArgumentException
// MODIFIES: this
// EFFECTS: If newSide is negative throws
// IllegalArgumentException; else update's
// this square's width
// and height to newSide.
public void setHeight(int newSide)
    throws IllegalArgumentException
// MODIFIES: this
// EFFECTS: If newSide is negative throws
// IllegalArgumentException; else update's
// this square's width
// and height to newSide.

```

考虑过程

```

static void scale(RectangleGeometry r, int sf) {
    r.setWidth(sf * r.getWidth());
    r.setHeight(sf * r.getHeight());
}

```

它按因子sf缩放矩形r的大小：RectangleGeometry对象作为参数调用scale过程时，一切工作正常。但是如果把一个SquareGeometry对象作为参数调用scale时，就会出现错误。为了找到原因，假设过程scale在如下代码段中使用，这段代码

调用构造器SquareGeometry产生一个边长为2的正方形，并希望把它放大2倍：

```
RectangleGeometry s = new SquareGeometry(2);
scale(s, 2);
System.out.println("width of s: " + s.getWidth());
System.out.println("height of s: " + s.getHeight());
```

代码段将输出什么结果？过程scale的调用是否完成了客户的要求？问题在哪里？类SquareGeometry是否满足替代原则？

5.6 Figure和Painter类

当考虑一个真实的对象，比如球时，你会认为外观是这个对象的必要部分。虽然一个红球有球属性和红色属性，当使用一个真实的球时，你认为它和它的所有属性是一个实体。扔一个红球，你不可能让它的球属性往一个方向飞，而它的红色属性往另一个方向飞。

在一节里，我们将设计一个Figure类，它把一个几何图形和它的外观结合起来。这样我们可以说：一个用蓝色填充的矩形为一个Figure对象。我们已经有了表示几何图形类型的Geometry接口。在本节中，将设计一个表示外观的Painter类型。Figure类型是由Painter和Geometry（参见图5-11）组成。

Painter接口定义如下：

```
public interface Painter {
    public abstract void paint(Graphics2D g2,
                               Geometry geometry);
    // REQUIRES: g2 and geometry are not null.
    // EFFECTS: Paints geometry into rendering context g2.
}
```

绘图工具（Painter）（实现Painter接口的对象）实现画图（paint）操作，以便画一个几何图形到绘图环境中。换句话说，绘图工具支持表达式

```
aPainter.paint(g2, aGeometry);
```

其中，g2是一个Graphics2D对象，paint方法实现的操作由aPainter引用对象定义，它可能是用特定的方法填充aGeometry，或只画出这个几何图形，或填充并画出这个几何图形，或者其他等等。

5.6.1 图形

在稍后我们将以Painter接口为根创建绘图工具类层次结构。这里先要介绍Figure类，见图5-11的类图。Figure为一个聚集，它最多包括一个Geometry和一个Painter。

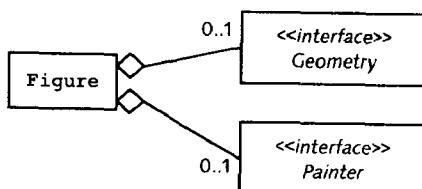


图5-11 图形组合了几何图形和外观

`Figure`对象提供的服务十分简单。它有一个`geometry`特性和一个`painter`特性，这暗示了可以随时修改它的几何图形和外观。另外它还提供了`paint`方法，这个方法要求`Graphics2D`对象作为输入参数。当图形收到`paint`消息时，它会把`paint`消息发送给它的绘图工具，并把它的几何图形和`Graphics2D`对象作为参数传递过去。下面是`Figure`类的定义：

```
public class Figure {

    protected Painter painter;
    protected Geometry geometry;
    public Figure(Geometry geometry, Painter painter) {
        this.geometry = geometry;
        this.painter = painter;
    }

    public Figure(Geometry geometry) {
        this(geometry, null);
    }

    public Figure(Painter painter) {
        this(null, painter);
    }

    public Figure() {
        this(null, null);
    }

    // painter property
    public Painter getPainter() { return painter; }

    public void setPainter(Painter painter) {
        this.painter = painter;
    }

    // geometry property
    public Geometry getGeometry() { return geometry; }

    public void setGeometry(Geometry geometry) {
        this.geometry = geometry;
    }

    public void paint(Graphics2D g2) {
        // REQUIRES: g2 is not null.
        // EFFECTS: Paints this figure's geometry into g2
        // using this figure's painter if the geometry
        // and painter are non-null;
        // else does nothing.
        if ((painter != null) && (geometry != null))
            painter.paint(g2, geometry);
    }
}
```

通过依次发送`paint`消息，可以把一个集合中的图形都绘制出来。例如，下面的程序把一个数组中`Figure`对象依次输出到绘图环境`g2`中：

```
static void paintManyFigures(Graphics2D g2,
    Figure[] figs) {
    for (int i = 0; i < figs.length; i++)
        figs[i].paint(g2);
}
```

由于多态性，不管`figs[i]`引用的几何图形和绘图工具的实际类型是什么，`paintManyFigure`方法都可以正常工作。

5.6.2 填充和画图的绘图工具

在创建某些图形之前我们需要定义某些实现`Painter`接口的具体类，再看一下`Painter`接口的定义：

```
public interface Painter {
    public abstract void paint(Graphics2D g2,
                               Geometry geometry);
    // REQUIRES: g2 and geometry are not null.
    // EFFECTS: Paints geometry into rendering context g2.
}
```

图5-12描述了本章余下部分将要讨论的`Painter`类的类图。

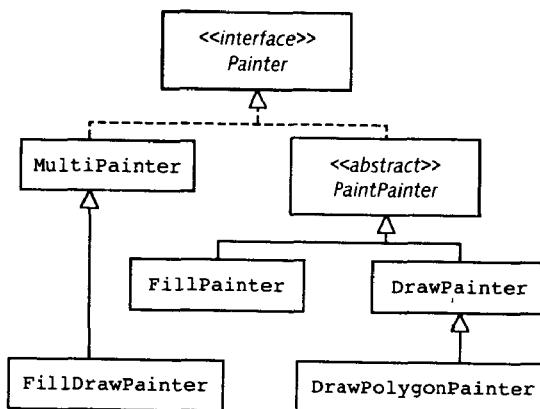


图5-12 Painter接口的子型

无论`Painter`要画一个图形轮廓还是填充内部区域都必须先设置好当前`Paint`（回想3.4节中`java.awt.Paint`接口描述了颜色模式被用于画图或填充操作，`Color`类实现了`Paint`）。为了子类使用方便，抽象类`paintPainter`定义了一个`paint`特性。下面是类`paintPainter`的类定义：

```
public abstract class PaintPainter implements Painter {
    protected Paint paint;
    protected static final Paint DefaultPaint = Color.white;
    protected PaintPainter(Paint paint)
        throws NullPointerException {
        setPaint(paint);
    }
    protected PaintPainter() {
        this(DefaultPaint);
    }
    // paint property
    public void setPaint(Paint newPaint)
        throws NullPointerException {
    }
}
```

```

        if (newPaint == null)
            throw new NullPointerException();
        this.paint = newPaint;
    }

    public Paint getPaint() {
        return this.paint;
    }
}

```

FillPainter类是用来填充几何图形的内部区域。它的paint（从父类**PaintPainter**继承而来）特性的当前值决定了当前使用的填充对象。当调用**FillPainter**的paint方法时，传入**Graphics2D**对象g2和**Geometry**对象为参数，它先设置绘图环境的paint属性，接着填充几何图形的内部区域。下面是**FillPainter**的类定义：

```

public class FillPainter extends PaintPainter {

    public FillPainter(Paint paint)
        throws NullPointerException {
        super(paint);
    }

    public FillPainter() {
        super();
    }

    public void paint(Graphics2D g2, Geometry geometry) {
        g2.setPaint(getPaint());
        g2.fill(geometry.shape());
    }
}

```

下面来用一个过程说明几何图形和绘图工具是如何结合成图形的。此过程有两个参数：一个点数组和一个Paint对象。它返回一个图形，这个图形是由输入点指定顶点的填充多边形：

```

static Figure makePolygonFigure(PointGeometry[] vertices,
                                Paint paint) {
    PolygonGeometry geometry = new PolygonGeometry(vertices);
    FillPainter painter = new FillPainter(paint);
    return new Figure(geometry, painter);
}

```

接下来将定义**DrawPainter**类，这个类也扩展了抽象类**PaintPainter**。**DrawPainter**类用来画几何图形的外形轮廓。同样从父类继承的paint特性的当前值决定当前使用的填充对象。**DrawPainter**还有一个stroke特性，使用它绘制几何图形外形。当调用**DrawPainter**的paint方法时，传入**Graphics2D**对象g2和**Geometry**对象为参数，它先设置g2的paint属性和stroke属性值，接着绘制几何图形的外形轮廓。下边是这个类的描述：

```

public class DrawPainter extends PaintPainter {

    public DrawPainter(Paint paint, Stroke stroke)
        throws NullPointerException
    // EFFECTS: If paint or stroke is null throws
    // NullPointerException; else initializes this
    // with the specified paint and stroke.
}

```

```

public DrawPainter(Paint paint)
    throws NullPointerException
// EFFECTS: If paint is null throws
//   NullPointerException; else initializes this
//   with the specified paint and default stroke
//   (one-pixel wide).

public DrawPainter()
// EFFECTS: Initializes this with the default
//   paint (white) and default stroke (1-pixel wide).
// stroke property
public void setStroke(Stroke newStroke)
    throws NullPointerException
// MODIFIES: this
// EFFECTS: If stroke is null throws
//   NullPointerException; else sets stroke
//   to newStroke.

public Stroke getStroke()
// EFFECTS: Returns the current stroke.

public void paint(Graphics2D g2, Geometry geometry)
// REQUIRES: g2 and geometry are not null.
// EFFECTS: Paints geometry into rendering
//   context g2 by drawing the outline using
//   the current paint and stroke.
}

```

练习

5.28 实现DrawPainter类。

5.29 编写一个图形程序，要求用红色填充多边形。程序的参数是图形顶点的坐标，与练习5.11中PaintPolygon程序定义相同。下边是执行命令行：

```
> java PaintPolygonWithFillPainter x0 y0 x1 y1 ...
```

必须用Figure对象表示多边形，其中几何图形是PolyGeometry对象，绘图工具是FillPainter对象。

5.30 编写一个图形程序，用蓝线画多边形的外形。该程序需要这样调用：

```
> java PaintPolygonWithDrawPainter stroke x0 y0 x1 y1 ...
```

程序的第一个参数是一个正整数，它决定笔的宽度，其他参数用来确定多边形的顶点。

5.31 使用Figure对象修改程序PaintManyRectangles的实现（4.2.6节）。程序中定义一个Figure对象数组或向量，代替原来的RectangleGeometry对象数组。每个Figure对象都有一个矩形几何图形和一个随机颜色的填充画笔。

5.6.3 组合绘图工具

到目前为止，我们已经实现的两个具体的绘图工具：FillPainter和DrawPainter，它们分别用来填充几何图形和绘画几何图形外形轮廓。如果我们想为给定几何图形既填充又画轮廓怎么办呢？这可以通过下面将要讨论的MultiPainter类来实现。

MultiPainter类有两个特性，分别是第一个绘图工具和第二个绘图工具，它们都是Painter类对象。当MultiPainter收到一个paint消息时，它首先把paint消息发送给第

一个绘图工具，接着在发送给第二个绘图工具。例如，下面的程序画一个矩形，其中矩形内部用绿色填充，轮廓用4像素宽的蓝色画出：

```
static void paintPrettyRectangle(Graphics2D g2) {
    Painter fill = new FillPainter(Color.green);
    Painter draw =
        new DrawPainter(Color.blue, new BasicStroke(4));
    Painter multiPainter = new MultiPainter(fill, draw);
    Geometry rectangle =
        new RectangleGeometry(0, 0, 100 100);
    Figure fig = new Figure(rectangle, multiPainter);
    fig.paint(g2);
}
```

类MultiPainter是一个聚集类，它的两个组成部分都是绘图工具。下面是类MultiPainter的定义：

```
public class MultiPainter implements Painter {

    protected Painter first, second;

    public MultiPainter(Painter first, Painter second)
        throws NullPointerException {
        // EFFECTS: If first or second is null throws
        // NullPointerException; else sets the first
        // painter and second painter.
        setFirstPainter(first);
        setSecondPainter(second);
    }

    public void paint(Graphics2D g2, Geometry geometry) {
        // REQUIRES: g2 and geometry are not null.
        // EFFECTS: Paints geometry into rendering context
        // g2 using the first painter followed by the
        // second painter.
        first.paint(g2, geometry);
        second.paint(g2, geometry);
    }

    public void setFirstPainter(Painter painter)
        throws NullPointerException {
        // MODIFIES: this
        // EFFECTS: If painter is null throws
        // NullPointerException; else sets the
        // first painter to painter.
        if (painter == null)
            throw new NullPointerException();
        first = painter;
    }

    public Painter getFirstPainter() {
        // EFFECTS: Returns the first painter.
        return first;
    }

    public void setSecondPainter(Painter painter)
        throws NullPointerException {
        // MODIFIES: this
        // EFFECTS: If painter is null throws
        // NullPointerException; else sets the
        // second painter to painter.
        if (painter == null)
```

```

        throw new NullPointerException();
        second = painter;
    }

    public Painter getSecondPainter() {
        // EFFECTS: Returns the second painter.
        return second;
    }
}

```

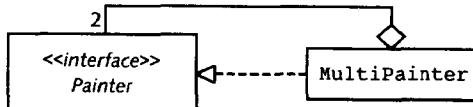


图5-13 由两个绘图工具组成的MultiPainter

利用MultiPainter类可以组合任意多的绘图工具。组合多于两个绘图工具时，可以把MultiPainter对象a的第二个绘图工具设置成另外一个MultiPainter对象b。当对象a收到一个paint消息时，它首先送一个paint消息给第一个绘图工具，接着再发送这个消息给第二个绘图工具b。作为响应，对象b发送一个paint消息给它自己的第一个绘图工具，接下来再发送给它自己的第二个绘图工具。很有可能对象b的第二个绘图工具也是一个MultiPainter对象。通过这种方式，可以创建一个连续的MultiPainter对象链：每个对象的第二个绘图工具都指向链中的下一个MultiPainter对象。无论何时链首的MultiPainter对象发送一个paint消息，paint消息都会在链中向下传递。例如，下面的程序绘制了一个矩形：用绿色填充矩形内部，用4像素宽的蓝线画外形轮廓，然后再用2个像素宽的红线在外形轮廓上面再画一次。

```

static void paintCoolRectangle(Graphics2D g2) {
    Painter fill = new FillPainter(Color.green);
    Painter drawBlue =
        new DrawPainter(Color.blue, new BasicStroke(4));
    Painter drawRed =
        new DrawPainter(Color.red, new BasicStroke(2));
    Painter multiDraw = new MultiPainter(drawBlue, drawRed);
    Painter multiPainter =
        new MultiPainter(fill, multiDraw);
    Geometry rectangle = new Rectangle(0, 0, 100 100);
    Figure fig = new Figure(rectangle, multiPainter);
    fig.paint(g2);
}

```

在程序最后，为响应paint消息，fig发送一个paint消息给MultiPainter。这个消息首先发送给绘图工具fill，它用绿色填充矩形；接着发送给绘图工具drawBlue，用4个像素宽的蓝线画外形轮廓；最后是发送给绘图工具drawRed，用2个像素宽的红线再画外形轮廓。

练习

5.32 在本节所给的过程paintPrettyRectangle中，假设要用下面语句替换其中第三个语句：

```
Painter multiPainter = new MultiPainter(draw, fill);
```

将会怎样影响图形的结果？

5.33 编写一个图形程序，要求用红色填充多边形，并且用蓝色画外形轮廓。程序用如下方法调用：

```
> java PaintPolygonWithPainter stroke x0 y0 x1 y1 ...
```

程序参数解释和练习5.30中一样。程序必须用一个Figure对象来描述多边形，其中Figure对象的几何图形是PolygonGeometry对象，绘图工具是MultiPainter对象。

5.34 (重点) 我们经常会需要这样的绘图工具：它首先填充一个几何图形，接着画几何图形的外形轮廓。FillDrawPainter类就具有这样的功能。请完成下面的实现：

```
public class FillDrawPainter
    extends MultiPainter {
    public FillDrawPainter(Paint fillPaint,
                           Paint drawPaint, Stroke stroke)
        throws NullPointerException {
        // EFFECTS: If fillPaint, drawPaint or stroke
        // is null throws NullPointerException; else
        // initializes this to fill with fillPaint,
        // then draw with drawPaint using stroke.
        ...
    }

    public FillDrawPainter(Paint fillPaint,
                           Paint drawPaint)
        throws NullPointerException {
        // EFFECTS: If fillPaint or drawPaint is null
        // throws NullPointerException; else
        // initializes this to fill with fillPaint,
        // then draw with drawPaint using a
        // 1-pixel-wide stroke.
        ...
    }
}
```

5.35 请描述下列类完成什么功能：

```
public class MultiDrawPainter implements Painter {
    protected Painter painter;

    public MultiDrawPainter(Paint[] paints)
        throws NullPointerException {
        painter = makePainter(paints, paints.length-1);
    }

    protected Painter makePainter(Paint[] paints,
                                  int n) {
        if (n == 0)
            return new DrawPainter(paints[0],
                                   new BasicStroke(2));
        else {
            Painter second = makePainter(paints, n - 1);
            Painter first = new DrawPainter(paints[n],
                                             new BasicStroke(2*(n + 1)));
            return new MultiPainter(first, second);
        }
    }

    public void paint(Graphics2D g2, Geometry geometry) {
```

```

    painter.paint(g2, geometry);
}
}

```

程序中使用了一个MultiDrawPainter类，它用程序的参数作为顶点画多边形：

```
> java DrawPolygonWithColors x0 y0 x1 y1 ...
```

多边形的外形轮廓用一条彩线画出：在8像素宽的红线上画6像素宽的绿线，接着再画4像素宽的蓝线，最后画2像素宽的黄线。

5.6.4 多边形绘图工具

到目前为止，PolygonGeometry对象的计算机绘图在表示单顶点多边形（1-gon画出一个点）和双顶点多边形（2-gon画出一条直线）时还不很完善。我们希望是像图5-5前两个图那样：1-gon画成一个回路，2-gon画成两条不同的曲线。这里设计DrawPolygonPainter类就是用来完成这个任务的。表达式：

```
aDrawPolygonPainter.paint(g2, aPolygonGeometry)
```

就是用所述方式画一个aPolygonGeometry多边形到绘图环境g2中。

类DrawPolygonPainter扩展DrawPainter类。它的构造器同父类的很像，其定义如下：

```

public DrawPolygonPainter(Paint paint, Stroke stroke)
    throws NullPointerException {
    super(paint, stroke);
}

public DrawPolygonPainter(Paint paint)
    throws NullPointerException {
    super(paint);
}
public DrawPolygonPainter() {
    super();
}

```

DrawPolygonPainter的paint方法实现和它父类的paint方法相似：它们都是先设置paint和stroke，接着再画几何图形的形状。惟一不同的是：如果图形是PolygonGeometry，用它的自己保护型的polygonShape方法创建多边形的形状。paint方法定义如下：

```

// method of DrawPolygonPainter class
public void paint(Graphics2D g2, Geometry geometry) {
    g2.setPaint(getPaint());
    g2.setStroke(getStroke());
    if (geometry instanceof PolygonGeometry)
        g2.draw(polygonShape((PolygonGeometry)geometry));
    else
        g2.draw(geometry.shape());
}

```

产生多边形形状的任务由polygonShape方法完成。需要定义了三个常量来控制1-gon和2-gon中曲线的尺寸：

```

// constants of DrawPolygonPainter class
// 1-gon case: controls length of loop:
//    larger values produce longer loops.

```

```

protected static final int LoopExtent = 40;
// 1-gon case: ratio of the loop's width to height
protected static final float LoopScale = 0.5f;
// 2-gon case: ratio of the loop's width to length
protected static final float TwoEdgeScale = 0.25f;

```

下面看**polygonShape**方法的处理：如果输入多边形至少有三个顶点，它的任务非常容易，把创建形状的任务交给多边形自身就可以了；如果少于三个顶点，则按要求绘制曲线创建形状。下面是这个方法的定义：

```

// method of DrawPolygonPainter class
protected Shape polygonShape(PolygonGeometry poly) {
    if (poly.nbrVertices() > 2)
        return poly.shape();
    else {
        GeneralPath path = new GeneralPath();
        PointGeometry v = poly.getVertex(0);
        path.moveTo(v.getX(), v.getY());
        if (poly.nbrVertices() == 1) { // poly is a 1-gon
            TransformablePointGeometry p =
                new TransformablePointGeometry(v);
            TransformablePointGeometry q =
                new TransformablePointGeometry(p);
            p.translate(LoopExtent,(int)(LoopExtent*LoopScale));
            q.translate(LoopExtent,(int)(-LoopExtent*LoopScale));
            path.curveTo(p.getX(), p.getY(), q.getX(), q.getY(),
                         v.getX(), v.getY());
        } else { // poly is a 2-gon
            PointGeometry v0 = poly.getVertex(0);
            PointGeometry v1 = poly.getVertex(1);
            int mX = (v1.getX() + v0.getX()) / 2;
            int mY = (v1.getY() + v0.getY()) / 2;
            TransformablePointGeometry m =
                new TransformablePointGeometry(mX, mY);
            int nX = (int)(-m.getY() * TwoEdgeScale);
            int nY = (int)(m.getX() * TwoEdgeScale);
            PointGeometry normal = new PointGeometry(nX, nY);
            m.translate(normal.getX(), normal.getY());
            path.quadTo(m.getX(), m.getY(), v1.getX(), v1.getY());
            m.translate(-2*normal.getX(), -2*normal.getY());
            path.quadTo(m.getX(), m.getY(), v0.getX(), v0.getY());
        }
        return path;
    }
}

```

只有一个顶点的多边形的情况下，**polygonShape**方法画一个由单个回路曲线构成的形状：两个控制点p和q引导这个曲线。**curveTo**方法从起始点v开始，加一条三次曲线到路径中，它有六个参数：前面四个为经过的控制点p和q的坐标，最后两个是曲线的终点坐标。终点坐标和起点v相等时（都为唯一的顶点坐标），就形成了一个回路。

在有两个顶点的情况下，**polygonShape**方法画两条曲线的：创建两个控制点，都在两个顶点连线的两边，控制点名为m，以它和v1为参数调用**quadTo**方法加一条二次曲线到路径中；移动m形成第二个控制点，然后以它和v0为参数调用**quadTo**方法加另一条二次曲线到路径中。

练习

5.36 修改练习5.33中**PaintPolygonWithPainter**程序，使用本节中方法画1-gon和

2-gon (如果多边形含有两个以上顶点，则把它描述成一个填充且绘出外形轮廓的多边形。)

5.37 修改练习5.35中的DrawPolygonWithColors程序，使用本节中方法画1-gon和2-gon。[提示：定义MultiDrawPainter类的子类：创建Multipainter对象使它能表示一系列的DrawPolygonPainter对象。]

5.38 编写一个命令交互程序：它可以绘画和编辑矩形图形（矩形或者椭圆），并输出到框架中。用户在创建图形时，可给每个图形起一个名字，且以后可用名字指定对应图形。程序包含三个特性：当前维数（位置、宽度和高度）、当前颜色和当前图形。用户用一个命令创建和命名一个新矩形：

```
new name rectangle
```

其中*name*是图形的名称。新的矩形*name*用当前维数初始化，并且用当前颜色填充。可用相似的方法创建和命名一个椭圆：

```
new name ellipse
```

允许修改已存在图形的特性。首先重新设置当前维数和当前颜色。例如，命令

```
color 0 255 255
dimensions 100 120 50 25
```

设置当前颜色为蓝绿色，设置当前维数为：位置（100, 120）、宽度50、高度25。接下来，用名字选择要修改的图形：

```
select name
```

最后，用不带参数的apply命令将当前设置应用到选定的图形：

```
apply
```

初始状态时，当前颜色是白色，当前图形为空，当前维数为：位置（0, 0）、宽度100、高度100。

程序支持下列命令：

- dimensions *x y width height* 设置当前维数为：位置（*x, y*）、宽*width*、高*height*。
- color *red green blue* 设置当前颜色，其中 $0 \leq red, green, blue \leq 255$ 。
- new *id* [rectangle | ellipse] 用当前维数和颜色创建一个新的图形，赋给它的名字为*id*。第二个参数决定图形是矩形还是椭圆。如果已经存在一个名叫*id*的图形，则新图形代替已存在图形。新的图形将成为当前图形。
- select *id* 把名字为*id*的图形设置为当前图形。如果没有名字叫*id*的图形，则什么也不做。
- apply 把当前维数和当前颜色设置应用到当前图形中。如果当前图形为空，则什么也不做。
- delete 删除当前图形，并把当前图形设置为空。如果当前图形为空，则什么也不做。
- print 输出当前维数、当前颜色、当前图形的名字、所有图形的名字。
- quit 退出程序。

为了响应每个命令，需要更新框架，以便显示图形的当前设置。当前图形（如

果有)为高亮度显示:用它的颜色填充内部,用线画出它的外形轮廓。其中外形轮廓必须是4个像素宽的白线上画2个像素宽的红线。其他图形用各自的颜色填充一下就可以了。

在下面的简单交互中,注释表明每条命令的效果:

```
> java PlayRectangularFigures
? color 255 0          // sets current color to red
? new a rectangle        // creates red rectangle a:(0,0),100,100
? dimensions 100 120 50 60 // sets dimensions to (100,120),50,60
? color 0 0 255         // sets current color to blue
? new b ellipse          // creates blue ellipse b:(100,120),50,60
? print
dimensions: (100,120),50,60
color: 0,0,255
current figure: b
figures: a b
? color 255 255 0      // sets current color to yellow
? dimensions 40 40 20 30 // sets dimensions to (40,40),20,30
? select a              // makes a the current figure
? apply                  // changes the color of a to yellow and
                           // the dimensions of a to (40,40),20,30
? quit
```

小结

像组合一样,继承是软件重用的关键机制。当使用继承来定义新类时,新类获得了已有类的域和方法。已有类称为超类或父类;新类称为子类。同样,子类也可以是它的子类的父类,这样就组成类继承层次结构。在类继承层次结构中,根类是所有类的父类。与此类似,接口也是实现它的类或扩展它的接口的父型。

多数情况下,类的实例是它父类所表达概念中的一种。例如,服务生是雇员的一种。然而,继承是一种多功能机制,通过它可以达到几个目的:

- 扩展继承:子类在继承的基础上增加新的域和方法。
- 特化继承:子类覆盖了属于父类的一个或多个方法。
- 说明继承:子类实现一个或多个父类指定却没有实现的抽象方法。类也可以实现任何由该类实现的接口指定的方法。

如果一个类实现了父类中所有的抽象方法,则这个类是具体类,否则这个类是抽象的。具体类可以被实例化,而抽象类只能被子类继承。类(具体类,抽象类)和接口都描述了一个接口,但是它们在实现方面有不同之处:具体类提供接口的完全实现;抽象类提供了一部分实现;接口则没有提供任何实现。一个类只能扩展一个父类,但是它可以实现多个接口。

在多态性的支持下,在要求父型对象的任何地方,可以使用子型对象来代替。也就是说:父型决定接口,对象的真实类型决定实际行为。替代原则规定:在不影响客户代码正确性的前提下,在任何要求父型出现的地方都可以使用子型对象。如果类和它的父型要求的说明保持一致,则这个类就满足了替代原则。

第6章 设计模式

由前面知识可以了解到，软件重用是面向对象技术中的一个优点。到目前为止，我们学习了两种重要的单个类的重用形式——组合和继承。下面我们将讨论另一种重要的重用形式——设计模式（design pattern），描述类和对象的排列以及它们之间的关系和协作。在本章中，先介绍设计模式的重要性；然后介绍了三种设计模式：迭代器设计模式、模板方法设计模式、组合设计模式；最后通过这三种模式和其他的模式讨论，介绍公认的模式分类的方法。

6.1 设计模式的重要性

设计模式是用于解决软件设计重复问题，就是说以前解决过的软件设计问题在下一次遇到时不应该再从头开始做，而是要利用以前的方法来解决。设计模式不仅描述常见的软件重用问题，而且展示如何解决这些问题的方法。这些方法灵活易用，可用在不同的情况下，并尽可能适用它所解决问题的各种可能的不同形式。

了解设计模式有几个目的：如果一个软件开发人员在遇到一个问题时，他可以查找是不是已有设计模式可以帮助解决该问题，如果找到了，他可以利用熟悉的设计模式很快地完成工作；同时他可以用设计模式将系统文档化，这样其他人可以很容易理解他的系统设计和意向。程序员学习设计模式不仅可以提高他的设计水平，还可以帮助理解已有的解决方法。此外，由于设计模式的定义和它的主要方法都被命名，这样大家可以使用这些术语思考或和其他人讨论设计问题。

设计模式的效用主要取决于它们是不是被恰当地描述。这是因为它既要尽可能抽象地表达所解决的问题，又要具体地描述在实际应用中如何使用它们。在《*Design Patterns: Elements of Reusable Object-Oriented Software*》（《设计模式：可重用面向对象软件基础》）一书中，提出一种描述设计模式的格式，书的作者是Gamma、Helm、Johnson和Vlissides。这是第一本系统讨论设计模式分类的书，书中介绍了设计模式的四个基本要素：

- **模式名：**标识模式。模式名描述设计模式并扩展设计模式词汇表。
- **问题：**描述模式何时可以使用。
- **解决方法：**定义设计模式中基本组成元素（类、接口、对象和方法）并说明每个元素的职责和它们之间的关系和协作；用一般术语和具体实例来表达解决方法。
- **结果：**描述使用模式所产生的结果和对比。在有多个可选模式时，可以帮助你选择最好的解决方法。

在6.2节到6.4节中，从设计模式的实用性和它们在设计图形程序中的具体作用出发，我们选择介绍了三种设计模式，它们是迭代器模式、模板方法模式、组合模式。迭代器模式（iterator pattern）用于访问一个聚集（如矢量、列表、多边形）中的各个元素，而又不暴露聚集接口或内部结构。模板方法模式（template method pattern）用于实现一个算法，其中的一些算法步骤可以不同。组合模式（composite pattern）用于创建表示复杂对象和组

合对象之间的层次结构，复杂对象是由相对简单的对象组成的。在6.5节中介绍了四种其他设计模式。这里所讨论的设计模式仅仅是现有模式的一小部分，在上述《*Design Patterns*》一书中，共描述了23种基本设计模式。

6.2 迭代器设计模式

迭代器设计模式用于访问一个聚集（有结构的一组元素如矢量、列表、字典、多边形）中的各个元素，而又不暴露聚集的接口或内部结构。迭代器对象给出一组顺序访问聚集元素的操作（通常称为遍历（traversing）聚集），并记录当前访问的元素位置，也称作迭代器的遍历状态（traversal state）。例如，一个列表的迭代器提供对列表中元素从头至尾的访问，并记录最后一次访问的元素的位置。列表迭代器提供的访问操作有：在未达到列表最后一个元素前访问列表中下一个元素的操作；测试是否到最后一个元素的操作。

迭代器模式负责访问和遍历聚集对象并将其放到迭代器对象中。通过使用迭代器，客户可以从处理聚集对象接口的复杂性中解脱出来，并且用更简单的迭代器的接口作为替代，其目的是限制对元素的访问。迭代器支持控制抽象（control abstraction），即为了控制对聚集元素的访问而对它的操作进行抽象。

在本节中，我们先看一个Java `Iterator`接口提供的这种设计模式的例子，然后设计的一个能遍历和修改多边型的迭代器。

6.2.1 Java的Iterator接口

请看下面的打印矢量中元素的过程`printVector`：

```
static void printVector(Vector a) {
    for (int i = 0; i < a.size(); i++) {
        Object obj = a.get(i);
        System.out.print(obj + " ");
    }
}
```

打印数组中元素的过程可以使用与`printVector`同样的控制逻辑。两个过程的差别只在于访问数组与矢量的不同。我们可以实现`printArray`过程如下：

```
static void printArray(Object[] a) {
    for (int i = 0; i < a.length; i++) {
        Object obj = a[i];
        System.out.print(obj + " ");
    }
}
```

过程`printVector`和`printArray`的区别在于对元素的访问的语法，两个过程使用同样控制逻辑：如果集合`a`有下一个元素，取出它并打印。实际上，这种控制逻辑可用于输出任何其他集合对象类型。

控制抽象抓住了访问和遍历聚集元素的控制逻辑，而隐藏了集合的类型的不同。迭代器的使用支持了控制抽象的实现。java.util包中包括多个迭代器接口，目前我们只讨论`java.util.Iterator`接口：

```
public interface Iterator {
    public boolean hasNext();
    // EFFECTS: Returns true if this iterator has more
    // elements; else returns false.
```

```

public Object next() throws NoSuchElementException;
// MODIFIES: this
// EFFECTS: If the iterator has no more elements
// throws NoSuchElementException; else returns
// the next element.

public void remove()
    throws UnsupportedOperationException,
    IllegalStateException;
// MODIFIES: this, and the underlying collection
// EFFECTS: If remove is not supported throws
// UnsupportedOperationException; else if next
// has not yet been called or remove has been
// called since the last call to next throws
// IllegalStateException; else removes the last
// element returned by this iterator.
}

```

`Iterator`对象是和一个它所遍历的基础集合（underlying collection）相关联的。`next`方法访问和返回集合中的下一个元素；如果集合中还有元素没有被访问，`hasNext`方法返回`true`；通常情况下，`next`方法调用前要先调用`hasNext`判断是否还有未访问元素。`next`方法一般仅在仍有没有被访问的元素时调用，习惯上，`hasNext`和`next`方法一起使用如下：

```
while (iter.hasNext())
    doSomething(iter.next());
```

其中`iter`实现了`Iterator`接口。

通过调用矢量的`iterator`方法可以获得一个矢量的`Iterator`对象。其中心是一个矢量对象，表达式

```
Iterator iter = v.iterator();
```

捕获了`v`的一个迭代器。这是因为`Vector`类实现了`java.util.Collection`接口，而这个接口的说明中包括`iterator`方法来获得`Iterator`对象。实际上只要是实现了`Collection`接口的任何对象`aCollection`，表达式

```
aCollection.iterator()
```

都返回一个`aCollection`的`Iterator`对象。

因为可能获得任何集合迭代器，所以我们可以设计一个使用`printVector`控制逻辑的过程，它能打印任何实现`Collection`接口的对象（包括`Vector`s）的元素。为了打印一个集合中的元素，首先获得集合的一个迭代器对象，然后使用迭代器的`hasNext`和`next`方法取得并打印其中的元素。过程如下：

```

static void printCollection(Collection c) {
    Iterator iter = c.iterator();
    while (iter.hasNext())
        System.out.print(iter.next() + " ");
}

```

这样，用`printCollection`过程打印`Vector`对象`v`中元素的调用方法如下：

```
printCollection(v);
```

打印数组元素的情况稍微复杂一些，因为数组没有实现`Collection`接口。但是可以将数

组看作是实现Collection接口的List，因此下面的调用打印数组a的元素：

```
printCollection(Arrays.asList(a));
```

需要说明的是，Iterator接口中的remove方法是用于从基础集合中删除next访问的最后一个元素的。实际上通常是不支持remove操作的，这样调用它时仅仅抛出一个异常。

练习

6.1 printCollection过程使我们能打印任何集合中的元素，如果我们要以另一种操作访问元素。这时需要另写一个过程。例如，假如要输出图形集合中的每个图形到给定的绘图上下文，则需要使用下面的过程：

```
static void paintCollection(Graphics2D g2, Collection c) {
    Iterator iter = c.iterator();
    while (iter.hasNext()) {
        Figure fig = (Figure)iter.next();
        fig.paint(g2);
    }
}
```

显然paintCollection和printCollection有相同的控制逻辑，不同的只是对集合中的每个元素的操作。使控制逻辑和它所用的操作脱离联系的一种好方法是把操作表示为对象。代表一个操作的对象称为函数符（functor）。函数符作为过程的参数传入到实现控制逻辑的过程，过程中又使用函数符代表的操作。通过定义不同的函数符，控制逻辑可以以不同的操作被重用。下面过程mapCollection的第一个参数map就是函数符，它定义的方法f代表期望的操作：

```
static void mapCollection(Functor map, Collection c) {
    Iterator iter = c.iterator();
    while (iter.hasNext())
        map.f(iter.next());
}
```

mapCollection过程将map.f操作应用于集合c中的每一个元素。其中Functor是一个接口，它的方法f可以对输入参数进行任何操作：

```
public interface Functor {
    public void f(Object obj);
    // MODIFIES: anything
    // EFFECTS: Any.
}
```

要使用mapCollection过程，先要定义一个类实现Functor接口。方法f的类实现决定了应用到集合中的元素上的操作。例如，要实现打印集合中的元素，可定义下面函数符类PrintFunctor：

```
public class PrintFunctor implements Functor {
    public void f(Object obj) {
        System.out.print(obj + " ");
    }
}
```

使用下面的语句我们可以打印Collection对象中的元素：

```
mapCollection(new PrintFunctor(), c);
```

作为另一个例子，我们可以定义下面的函数符，类PaintFunctor把Figure的对象集合绘制到给定的绘图环境：

```
public class PaintFunctor implements Functor {
    protected Graphics2D g2;

    public PaintFunctor(Graphics2D g2) {
        this.g2 = g2;
    }

    public void f(Object obj) {
        Figure fig = (Figure)obj;
        fig.paint(g2);
    }
}
```

然后我们可以使用下面的语句把对象集合figs绘制到绘图环境g2：

```
mapCollection(new PaintFunctor(g2), figs);
```

(a) 使用PaintFunctor重写练习5.31中的PaintManyRectangles类的paintComponent方法。

(b) 定义一个新的函数符类AttachPainterFunctor，它能把一个Painter对象传给到Figure集合中的每个元素。AttachPainterFunctor的构造器的参数为任何一个实现Painter接口的对象，下面的表达式附加绘图工具aPainter到集合figs中的每个Figure：

```
mapCollection(new AttachPainterFunctor(aPainter),
    figs);
```

(c) 定义函数符类TranslateFunctor，它对集合中的每个几何图形，在x轴移动dx单位，y轴移动dy单位。通过TranslateFunctor的构造器传入参数dx和dy。下面是一个使用示例，在该例中集合geometries最初包含PointGeometry对象(1, 2)和(4, 5)，然后它们以dx=6和dy=-2被移动：

```
mapCollection(new PrintFunctor(), geometries);
// (1,2) (4,5)
mapCollection(new TranslateFunctor(6,-2),geometries);
mapCollection(new PrintFunctor(), geometries);
// (7,0) (10,3)
```

6.2.2 动态多边形

以前设计的polygonGeometry类的实例，在结构不随时间改变的条件下是静态的。相反，动态多边形是指提供插入新顶点、删除现存顶点、移动现存顶点移到新位置的操作。在本节中将使用迭代器模式来实现对多边形的访问和修改，为了使用动态多边形，客户需要获得多边形的迭代器，然后使用迭代器的操作处理多边形。

接口PolygonIterator说明由多边形迭代器完成的操作，它是由Dynamic-PolygonIterator类来实现的。在本节后面会给出它们的详细定义。图6-1表明一个动态多边形迭代器拥有一个动态多边形几何图形——迭代器对它的基础多边形进行操作，同集合迭代器对它的基础集合进行操作的情形非常相似。图6-1同时说明允许任意个多边形迭代器对同一基础多边形进行操作。

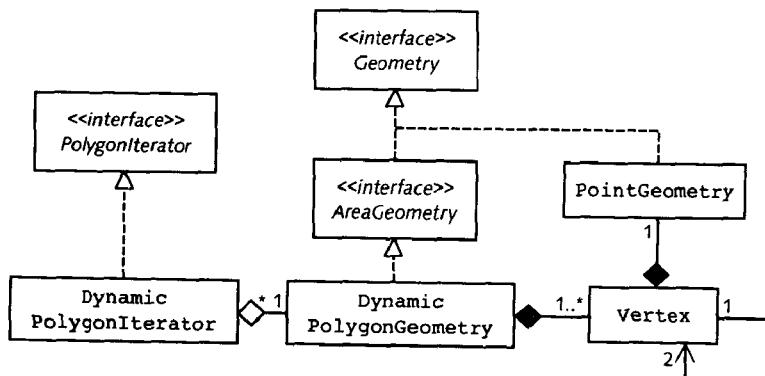


图6-1 动态多边形由一个或多个顶点组成，并且一个顶点引用一个点
(它的位置)和两个顶点(它的前身顶点和后继顶点)

图6-1同时展示了动态多边形的结构：一个动态多边形对象由一个或多个顶点(**Vertex**)对象组成，并按顶点对象出现的先后排序。一个顶点对象又为其在平面中的位置定义了一个域(**PointGeometry**对象)，并在多边形(顶点对象)内为它的前一个顶点和它的后一个顶点定义了两个域)。对这种设计后面还会有更详细的解释。

下面将先介绍动态多边形的具体行为，然后实现类**DynamicPolygonGeometry**和它依赖的**Vertex**类，最后介绍多边形迭代器的行为和它的实现。

1. 动态多边形的行为

动态多边形与静态多边形(由类**PolygonGeometry**实现)的区别主要表现在两个方面：第一，与静态多边形通过附标访问它的顶点和边的操作不同，动态多边形中提供访问迭代器的操作，用迭代器实现访问它的顶点和边。动态多边形由客户通过迭代器的方式控制。第二，提供一种保护型方法用于插入新顶点和删除现有顶点(**insertAfterVertex**和**removeVertex**)，这两种方法属于多边形的保护型接口，其意图在于**PolygonIterator**的子类更容易实现(通常客户没有权力访问这两个方法)。下面是**DynamicPolygonGeometry**的类框架：

```

public class DynamicPolygonGeometry
    implements AreaGeometry {

    public DynamicPolygonGeometry(PointGeometry p)
        throws NullPointerException
    // EFFECTS: If p is null throws NullPointerException;
    //   else constructs a one-vertex polygon positioned
    //   at p.

    public DynamicPolygonGeometry(PointGeometry[] points)
        throws NullPointerException, ZeroArraySizeException
    // EFFECTS: If points is null or points[i] is null
    //   for some legal i throws NullPointerException;
    //   else if points has length zero throws
    //   ZeroArraySizeException; else creates a polygon
    //   whose vertex sequence is given by points.

    public int nbrVertices()
    // EFFECTS: Returns the number of vertices in
    //   this polygon.
}

```

```

public PolygonIterator iterator()
    // EFFECTS: Returns a new iterator for this polygon.

public String toString()
    // EFFECTS: Returns "Polygon: v0, v1, ... vn-1"
    // where each vi describes vertex i.

    //
    // implements the Geometry and AreaGeometry interfaces
    //
public Shape shape()
    // EFFECTS: Returns the shape of this polygon.

public void translate(int dx, int dy)
    // MODIFIES: this
    // EFFECTS: Translates this polygon by dx and dy.

public boolean contains(int x, int y)
    // EFFECTS: Returns true if this polygon contains
    // the point (x,y); else returns false.

public boolean contains(PointGeometry p)
    throws NullPointerException
    // EFFECTS: If p is null throws NullPointerException;
    // else returns true if this polygon contains p;
    // else returns false.

protected Vertex vertex()
    // EFFECTS: Returns some vertex in this polygon.

protected Vertex insertAfterVertex(Vertex v,
    PointGeometry p)
    // REQUIRES: v and p are not null, and v belongs
    // to this polygon.
    // MODIFIES: this
    // EFFECTS: Inserts a new vertex w after v at
    // position p, and returns the new vertex w.

protected void removeVertex(Vertex v)
    // REQUIRES: v is not null, v belongs to this
    // polygon, and this polygon contains at least one
    // other vertex besides v.
    // MODIFIES: this
    // EFFECTS: Removes vertex v from this polygon.
}

```

DynamicPolygonGeometry类中最有趣的部分是它的保护型接口，它们的用法将在本节后面介绍多边形迭代器时讨论。现在我们只满足于一个简单的过程，它用DynamicPolygonGeometry类的保护型接口创建一个多边形。这个过程buildPolygon的调用参数为 $n(n > 0)$ 个点的数组，这些点确定了它新建和返回的 n 边形的 n 个顶点：

```

static DynamicPolygonGeometry
buildPolygon(PointGeometry[] points) {
    DynamicPolygonGeometry poly =
        new DynamicPolygonGeometry(points[0]);
    Vertex v = poly.vertex();
    for (int i = 1; i < points.length; i++)
        v = poly.insertAfterVertex(v, points[i]);
    return poly;
}

```

例如，代码段

```
PointGeometry[] points = { new PointGeometry(5, 5),
    new PointGeometry(20, 5),
    new PointGeometry(10, 15)
};
DynamicPolygonGeometry poly = buildPolygon(points);
```

将产生一个由点 (5, 5), (20, 5) 和 (10, 15) 组成的三边形，顶点是向前旋转的，即 (5, 5) 在 (20, 5) 之前，(20, 5) 又在 (10, 15) 之前。注意，`buildPolygon`过程中本来可以直接调用 `DynamicPolygonGeometry` 的第二个构造器完成整个多边形的创建工作，但是本例是为了示例保护型接口的行为，所以没有这样做。

练习

6.2 使用 `DynamicPolygonGeometry` 的第二个构造器重写 `buildPolygon` 过程。

2. Vertex类

下面着重来看动态多边形是如何表示的。按我们所采用的存储结构，多边形是由一组顶点 (`Vertex`) 对象组成，每个对象代表多边形的一个顶点。顶点对象之间又组成一个双向链表，即一个点对象保存有它的前一个和后一个点对象的引用，这样就可以进行双向（向前或向后）遍历。一个 `Vertex` 也存储一个 `PointGeometry` 对象，由它确定顶点在平面中的位置。

类 `DynamicPolygonGeometry` 定义了两个域：`vertex` 引用多边形的顶点；整型的 `nbrVertices` 表示多边形中顶点的数量。这对理解用于表示非退化的动态多边形的存储结构是有帮助的，图 6-2 就表示下面对象 `poly` 的存储结构：

```
PointGeometry[] points = { new PointGeometry(5, 5),
    new PointGeometry(20, 5),
    new PointGeometry(10, 15)
};
DynamicPolygonGeometry poly = buildPolygon(points);
```

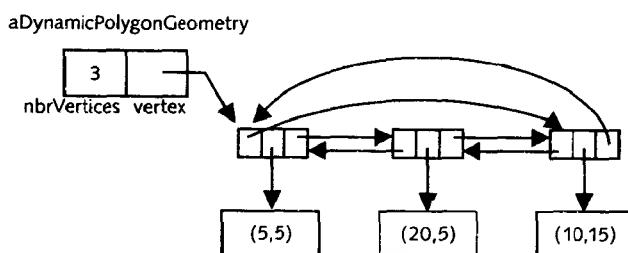


图 6-2 动态多边形的表示

图 6-2 中，箭头表示引用：箭头的起点为引用对象的存储位置，箭头所指为所引用的对象。需要注意的是：存储结构图并没有表明 `DynamicPolygonGeometry` 对象引用哪一个顶点，这就是说它有可能引用任意一个顶点。还应该注意的是：当只有一个顶点时，它会两次指向自己，因为它的前一个和后一个顶点都是它自己。下面是类 `Vertex` 的说明：

```
class Vertex {
    protected Vertex(PointGeometry p)
```

```

// REQUIRES: p is not null.
// EFFECTS: Initializes this vertex at location p,
// and makes it its own successor and predecessor.

protected PointGeometry point()
// EFFECTS: Returns a reference
// to this vertex's position.

protected Vertex next()
// EFFECTS: Returns a reference to
// this vertex's successor.

protected Vertex prev()
// EFFECTS: Returns a reference
// to this vertex's predecessor.

protected Vertex insertAfter(PointGeometry p)
// REQUIRES: p is not null.
// MODIFIES: this
// EFFECTS: Constructs a new vertex v at position p,
// makes v this vertex's successor and returns v.

protected void remove()
// MODIFIES: this
// EFFECTS: Removes this vertex, and makes this
// vertex's successor the successor of this
// vertex's predecessor.

public String toString()
// EFFECTS: Returns a string-descriptor for this
// vertex, indicating its position.
}

```

类Vertex的实现以下面三个域表示一个顶点：

```

// fields of Vertex class
// position of this vertex
protected PointGeometry point;
// next and previous vertices
protected Vertex next, prev;

```

类Vertex有一个单参数构造器，它产生新顶点作为它的前一个顶点和后一个顶点：

```

protected Vertex(PointGeometry point) {
    this.point = point;
    this.next = this;
    this.prev = this;
}

```

类Vertex提供访问其位置（一个点）以及它的前一个顶点和后一个顶点的方法point()、next()、prev()：

```

// methods of Vertex class
protected PointGeometry point() {
    return this.point;
}

protected Vertex next() {
    return this.next;
}

protected Vertex prev() {

```

```

    return this.prev;
}

```

顶点的串描述符与其位置的串描述符相同：

```

// method of Vertex class
public String toString() {
    return point.toString();
}

```

图6-3表示对顶点链表调用insertAfter和remove方法的结果，其中变量a引用一个顶点，指令

```
Vertex b = a.insertAfter(anyPoint);
```

将使图6-3从L变成R，变量b引用定位在anyPoint的新顶点。如果执行指令

```
b.remove();
```

将使b被取消，顶点链表从R恢复到L。

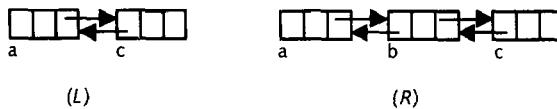


图6-3 对于L，执行a.insertAfter(aPoint)产生R；对于R，执行b.remove()产生L

insertAfter将依输入参数生成新顶点newV，然后把newV插入到当前顶点的后面。为了把新顶点链接到链表中，必须设置newV的两个链域，并更新newV的前一个顶点（即当前顶点）的next域和它的后一个顶点的prev域。下面是该方法的定义：

```

// method of Vertex class
protected Vertex insertAfter(PointGeometry p) {
    Vertex newV = new Vertex(new PointGeometry(p));
    Vertex prev = this;
    Vertex next = this.next();
    // link newV into the chain of vertices
    newV.prev = prev;
    newV.next = next;
    prev.next = next.prev = newV;
    return newV;
}

```

Vertex.remove方法的实现将作为练习完成。

练习

6.3 为什么Vertex类的toString方法的类型定义为公有的？如果toString定义为保护型，Java编译器会产生什么错误消息？

6.4 下面语句每执行一步，画出动态多边形poly的存储结构图：

```

DynamicPolygonGeometry poly =
    new DynamicPolygon(new PointGeometry(4, 5));
Vertex v = poly.vertex();
poly.insertAfterVertex(v, new PointGeometry(8, 9));
v = poly.insertAfterVertex(v, new PointGeometry(6, 7));
v = v.next();
poly.removeVertex(v);

```

6.5 实现Vertex.remove()方法。

3. 实现动态多边形

现在可以在6.2.2节开始时给出的类框架的基础上实现DynamicPolygonGeometry类。它有两个实例域：vertex域保存顶点链中的某个Vertex对象的引用，nbrVertices域保存顶点的数量。下面是完整的类定义：

```
public class DynamicPolygonGeometry
    implements AreaGeometry {

    protected Vertex vertex;
    protected int nbrVertices;

    public DynamicPolygonGeometry(PointGeometry point)
        throws NullPointerException {
        vertex = new Vertex(point);
        nbrVertices = 1;
    }

    public DynamicPolygonGeometry(PointGeometry[] points)
        throws NullPointerException, ZeroArraySizeException {
        if (points.length == 0)
            throw new ZeroArraySizeException();
        this.vertex = new Vertex(points[0]);
        this.nbrVertices = 1;
        Vertex v = this.vertex;
        for (int i = 1; i < points.length; i++)
            v = insertAfterVertex(v, points[i]);
    }

    public int nbrVertices() {
        return this.nbrVertices;
    }

    public PolygonIterator iterator() {
        return new DynamicPolygonIterator(this);
    }

    protected String toString() {
        return "dynamic polygon: " + verticesToString();
    }

    public Shape shape() {
        GeneralPath path = new GeneralPath();
        Vertex v = vertex();
        path.moveTo(v.point().getX(), v.point().getY());
        for (int i = 0; i < nbrVertices() - 1; i++) {
            v = v.next();
            path.lineTo(v.point().getX(), v.point().getY());
        }
        path.closePath();
        return path;
    }

    public void translate(int dx, int dy) {
        ...
    }

    public boolean contains(int x, int y) {
        return shape().intersects(x-0.01,y-0.01,.02,.02);
    }
}
```

```

public boolean contains(PointGeometry p) {
    return contains(p.getX(), p.getY());
}

protected Vertex vertex() {
    return this.vertex;
}

protected Vertex insertAfterVertex(Vertex v,
                                   PointGeometry newPoint) {
    ...
}

protected void removeVertex(Vertex v) {
    if (v == this.vertex)
        this.vertex = v.next();
    v.remove();
    --nbrVertices;
}

protected String verticesToString() {
    String res = "";
    Vertex v = vertex();
    for (int i = 0; i < nbrVertices - 1; i++) {
        res += v + ",";
        v = v.next();
    }
    res += v;
    return res;
}
}

```

其中`DynamicPolygon.iterator`方法返回一个`DynamicPolygonIterator`类的实例，它的定义将会出现在下一节中。

练习

6.6 完成`DynamicPolygonGeometry`类的定义。

6.2.3 多边形迭代器

动态多边形将使用多边形迭代器对象（实现`PolygonIterator`接口）来实现。使用多边形迭代器的客户可以自由地轮流访问多边形的顶点，增加、删除、移动多边形的顶点。客户可以通过发送`iterator`消息得到动态多边形的迭代器。

多边形迭代器在任何时候都定位在它的基本多边形的某个顶点，这个顶点称为迭代器的当前顶点（`current vertex`）。利用迭代器的`point`方法获得当前顶点在平面上的位置。随着遍历的进行，不同的顶点成为当前顶点。`next`和`prev`方法用来把迭代器从一个顶点移到另一个顶点：`next`方法把它移到当前顶点的后一个顶点，而`prev`方法把它移到当前顶点的前一个顶点。连续调用`next`方法会使迭代器顺序遍历顶点集，而连续调用`prev`是逆序遍历顶点集。迭代器的`edge`方法返回当前边（`current edge`），当前边是连接当前顶点和它的后一个顶点的边。

多边形迭代器的修改方法是对当前顶点进行的。`insertAfter`方法是在当前顶点后插入一个新的点，该方法的参数指出插入顶点的位置，调用此方法后新插入的顶点变成

当前顶点。`remove`方法删除当前顶点，并将当前顶点指向被删顶点的前一个顶点（如果顶点集中只有一个顶点，则调用`remove`方法会产生异常）。`moveTo`和`moveBy`方法用来把当前顶点移动到平面上的某一个位置。下面是`PolygonIterator`接口的定义：

```

public interface PolygonIterator {
    public PointGeometry point();
    // EFFECTS: Returns position of the current vertex

    public LineSegmentGeometry edge();
    // EFFECTS: Returns the current edge.

    public void next();
    // MODIFIES: this
    // EFFECTS: Moves this iterator to the current
    // vertex's successor.

    public void prev();
    // MODIFIES: this
    // EFFECTS: Moves this iterator to the current
    // vertex's predecessor.

    public void insertAfter(PointGeometry p)
        throws NullPointerException;
    // MODIFIES: this, and the underlying polygon
    // EFFECTS: If p is null throws NullPointerException;
    // else inserts a new vertex v, positioned at p,
    // as the successor to the current
    // vertex and makes v the current vertex.

    public void remove() throws IllegalStateException;
    // MODIFIES: this, and underlying polygon
    // EFFECTS: If nbrVertices()==1 throws
    // IllegalStateException; else removes the current
    // vertex and makes its predecessor the current
    // vertex.

    public void moveTo(int x, int y);
    // MODIFIES: underlying polygon
    // EFFECTS: Moves the current vertex to (x,y).

    public void moveBy(int dx, int dy);
    // MODIFIES: underlying polygon
    // EFFECTS: Translates the current vertex by dx
    // and dy.

    public int nbrVertices();
    // EFFECTS: Returns the number of vertices in the
    // underlying polygon.
}

```

1. 移动多边形

下面用两个例子说明如何使用多边形迭代器。第一个例子是移动多边形的一个过程，把多边形`poly`沿x轴移动`dx`单位，沿y轴移动`dy`单位。实现思想是把多边形的每个顶点都移动`dx`, `dy`。过程`translatePolygon`由参数`poly`获得迭代器对象，使用它访问多边形的全部顶点。在每个顶点处调用`moveBy`方法移动顶点，下面为程序：

```

static void translatePolygon(DynamicPolygonGeometry poly,
                            int dx, int dy) {

```

```

    PolygonIterator iter = poly.iterator();
    for (int i=0; i<iter.nbrVertices(); i++, iter.next())
        iter.moveBy(dx, dy);
}

```

for循环中条件 `i < iter.nbrVertices()` 正好使所有顶点被访问一次。

2. 建立规则多边形

第二个例子涉及建立规则多边形 (regular polygon)，这是最简单的多边形，即所有边和角都相等的多边形。例子中包括等边三角形和正方形。我们将写出这个过程 `buildRegularPolygon`，它的头定义如下：

```

static DynamicPolygonGeometry
buildRegularPolygon(int n, int rad,
                    PointGeometry center,
                    double twist)

```

过程返回 `n` 边规则多边形，多边形以点 `center` 为圆心，以 `rad` 为半径。多边形沿中心点顺时针旋转 `twist` 度。图 6-4 所示的规则五边形是调用下面语句产生的：

```
buildRegularPolygon(5, 1, new PointGeometry(2, 2), 45);
```

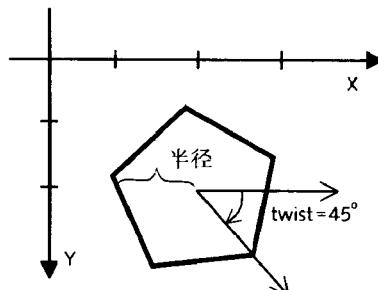


图 6-4 规则五边形

过程 `buildRegularPolygon` 要分三步实现：第一步，创建只有一个顶点 `center` 的多边形 `poly`，并获取 `poly` 的迭代器，顶点 `center` 是一个假顶点，最后会被删除。第二步，循环插入多边形的顶点。在每次循环中，先建立一个可移动的原点 `p`，沿正 `x` 轴从 `(rad, 0)` 移动点 `p`，然后按顺时针方向以当前旋转角旋转点 `p`，再在迭代器的当前顶点之后插入顶点 `p`。第三步，删除假顶点，返回多边形 `poly`。实现如下：

```

static DynamicPolygonGeometry
buildRegularPolygon(int n, int rad, PointGeometry center,
                    double twist)
throws NullPointerException {
    // step 1: build a new polygon and obtain iterator
    DynamicPolygonGeometry poly =
        new DynamicPolygonGeometry(center);
    PolygonIterator iter = poly.iterator();
    // step 2: insert the n vertices
    double twistInc = 360.0 / n;
    for (int i = 0; i < n; i++, twist += twistInc) {
        TransformablePointGeometry p =
            new TransformablePointGeometry();
        p.translate(rad, 0);
        p.rotate(twist);
        p.translate(center.getX(), center.getY());
        iter.insertAfter(p);
    }
}

```

```

    }
    // step 3: remove the dummy vertex
    iter.next();           // advance iter to dummy vertex
    iter.remove();         // and remove it
    return poly;
}

```

注意，在上面的实现过程中因为创建多边形时不能出现空多边形，所有第一步中的位于center的假顶点是必需有的。需要在完成真正的顶点插入后把假顶点删除。

3. 实现多边形迭代器

下面我将定义类DynamicPolygonIterator，它实现了PolygonIterator接口。它定义了两个实例域：polygon域存储迭代器的基础多边形的一个引用，vertex域保存迭代器的当前顶点。下面是实现程序：

```

public class DynamicPolygonIterator
    implements PolygonIterator {

    // the underlying polygon
    protected DynamicPolygonGeometry polygon;
    // the current vertex
    protected Vertex vertex;

    public DynamicPolygonIterator(DynamicPolygonGeometry poly)
        throws NullPointerException {
        // EFFECTS: If poly is null throws
        // NullPointerException; else constructs this for
        // the polygon poly and any current vertex.
        this.polygon = poly;
        this.vertex = poly.vertex();
    }

    public DynamicPolygonIterator(DynamicPolygonIterator iter)
        throws NullPointerException {
        // EFFECTS: If iter is null throws
        // NullPointerException; else constructs this for
        // iter's underlying polygon and current vertex.
        this.polygon = iter.polygon;
        this.vertex = iter.vertex();
    }

    public PointGeometry point() {
        return new PointGeometry(vertex.point());
    }

    public LineSegmentGeometry edge() {
        return new LineSegmentGeometry(vertex.point(),
                                      vertex.next().point());
    }

    public void next() {
        vertex = vertex.next();
    }

    public void prev() {
        vertex = vertex.prev();
    }

    public void insertAfter(PointGeometry p)
        throws NullPointerException {
        if (p == null) throw new NullPointerException();
    }
}

```

```

        vertex = polygon.insertAfterVertex(vertex, p);
    }

    public void remove() throws IllegalStateException {
        if (nbrVertices() == 1)
            throw new IllegalStateException();
        Vertex prevVertex = vertex.prev();
        polygon.removeVertex(vertex);
        vertex = prevVertex;
    }

    public void moveTo(int x, int y) {
        PointGeometry p = vertex.point();
        p.setX(x);
        p.setY(y);
    }

    public void moveBy(int dx, int dy) {
        PointGeometry p = new PointGeometry(vertex.point());
        moveTo(p.getX() + dx, p.getY() + dy);
    }

    public int nbrVertices() {
        return polygon.nbrVertices();
    }
}

```

由于类DynamicPolygonIterator中调用了类DynamicPolygonGeometry和类Vertex中的保护型接口，所以在使用时要求它们在同一个包中。Java中保护型接口就是这样使用的，所以说上面的实现没有破坏类的封装。

练习

- 6.7 试用多边形迭代器重新实现DynamicPolygonGeometry类的verticesToString方法和shape方法。
- 6.8 定义同5.6.4节中类DrawPolygonPainter（用于静态多边形）功能相似的类DrawDynamicPolygonPainter（用于动态多边形）。
- 6.9 以命名为DynamicPolygons的类的静态方法实现buildRegularPolygon过程，该类包含对多边形进行操作的方法，就像类java.util.Arrays包括各种对数组进行操作的方法一样。

编写图形程序PaintRegularPolygon，当用

```
> java PaintRegularPolygon n radius [twist]
```

调用时将产生一个规则多边形，它的中点在程序框架中心，按你选择的颜色画图和填充，半径为radius，旋转角为twist度（如果没有最后一个参数，则twist为0）。

- 6.10 星形多边形可以由规则多边形改造而成。改造方法如下：沿顶点和中心的连线方向移动间隔顶点（每隔一个移动一个），移动的幅度由移动系数tf决定。下面buildStarPolygon过程的前四个参数定义和buildRegularPolygon相同，第五个参数tf决定被选定的顶点移向或离开多边形的中心的距离。实际上参数tf决定星形多边形的凸凹程度。下面为buildStarPolygon过程：

```
public static DynamicPolygonGeometry
buildStarPolygon(int n, int rad, PointGeometry center,
```

```

        double twist, double tf)
    throws NullPointerException {
DynamicPolygonGeometry poly =
    buildRegularPolygon(n, radius, center, twist);
PolygonIterator iter = poly.iterator();
for (int i = 0; i < iter.nbrVertices() / 2; i++) {
    PointGeometry p = iter.point();
    int dx = (int)((center.getX() - p.getX()) * tf);
    int dy = (int)((center.getY() - p.getY()) * tf);
    iter.moveBy(dx, dy);
    iter.next();
    iter.next();
}
return poly;
}

```

把buildStarPolygon过程加入到类DynamicPolygons中，然后实现下面画星形多边形的图形程序PaintStarPolygon：

```
> java PaintStarPolygon n radius tf
```

试着改变tf的值，如tf大于1时，图形会发生何种变化？

6.11 练习6.9中，通过DynamicPolygons类的一个方法实现规则多边形。你也可以用另一个类RegularPolygonGeometry来实现规则多边形，这个类扩展了DynamicPolygonGeometry类。这个类用于创建一个新的规则多边形，并可以通过多边形迭代器（RegularPolygonGeometry类从它的父类继承Iterator方法）来操作维护它。下面是类框架：

```

public class RegularPolygonGeometry
    extends DynamicPolygonGeometry {
public RegularPolygonGeometry(int n, int radius,
    PointGeometry center, double twist)
    throws NullPointerException,
        IllegalArgumentException
// EFFECTS: If center is null throws
// NullPointerException; else if n<3 throws
// IllegalArgumentException; else constructs
// a regular n-gon of given center and twist.

public RegularPolygonGeometry(int n, int radius,
    PointGeometry center)
    throws NullPointerException,
        IllegalArgumentException
// EFFECTS: If center is null throws
// NullPointerException; else if n<3 throws
// IllegalArgumentException; else constructs
// a regular n-gon with given center and
// zero twist.

public RegularPolygonGeometry(int n, int radius)
    throws IllegalArgumentException
// EFFECTS: If n<3 throws
// IllegalArgumentException; else constructs
// a regular n-gon centered at the origin
// and with zero twist.
}

```

试着实现这个类。修改你的PaintRegularPolygon程序（练习6.9），使得用这个类

替代DynamicPloygon类的buildRegularPolygon方法时，就创建一个规则多边形。

6.12 编写一个交互程序PlayDynamicPolygon，供用户使用命令行创建和编辑动态多边形。你的程序需要保存多边形的当前顶点，大多数命令都要使用它。在任何时候都要显示多边形，并用高亮度显示当前顶点。程序开始执行时，没有多边形创建，但当用户插入一个顶点后，程序就创建了新的一个顶点的多边形。为便于说明，在下面的描述中将把没有多边形存在的状态称为空多边形（empty polygon）。下面是该程序支持的所有命令：

- **insert x y** 在当前顶点后插入一个新顶点（ x, y ）；新顶点成为当前顶点。如果现在为空多边形，则创建一个顶点（ x, y ）的多边形。
- **remove** 删除当前顶点并设置它的前一个顶点为当前顶点；如果没有顶点了，则变成空多边形。
- **moveto x y** 移动当前顶点到位置（ x, y ）；如果为空多边形，则不做任何操作。
- **translate dx dy** 把当前顶点 x 坐标移动 dx , y 坐标移动 dy ；如果为空多边形，则不做任何操作。
- **next** 将当前顶点的后一个顶点设置为当前顶点；如果为空多边形，则不做任何操作。
- **previous** 将当前顶点的前一个顶点设置为当前顶点；如果为空多边形，则不做任何操作。
- **clear** 清除多边形，即变成空多边形。
- **size** 打印出多边形的顶点数。
- **quit** 退出程序。

下面是程序运行示例：

```
> java PlayDynamicPolygon
? insert 10 10
? insert 50 50 // left diagram of Figure 6.5
? insert -25 50
? insert -50 0
? next          // middle diagram of Figure 6.5
? remove        // right diagram of Figure 6.5
? quit
```

图6-5显示了交互的三个阶段。当前顶点表示为一个黑点。坐标系间距为50单位。

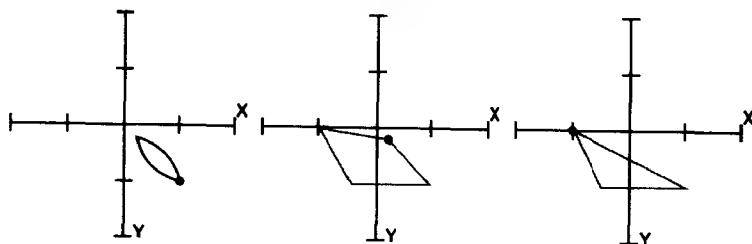


图6-5 动态多边形交互图

[提示：你的程序PlayDynamicPloygon使用MyInteractiveProgram模板实现。控制器类有一个保存多边形的域和多边形迭代器的域（多边形为空时，两个域都为null），同时提供返回当前多边形的方法和返回当前顶点位置的方法。这两个方法是为

PlayDynamicPolygon.paintComponent方法准备的。】

6.13 扩展PlayDynamicPolygonGeometry类，定义TriangleGeometry类用来创建三角形。这个类只有一个方法——一个构造器。构造器带有三个点参数，如果它们不在一条直线上，用它们作顶点构造一个动态三角形：

```
public TriangleGeometry(PointGeometry a,
                        PointGeometry b,
                        PointGeometry c)
    throws NullPointerException,
           ColinearPointsException
// EFFECTS: If a, b, or c are null throws
// NullPointerException; else if a, b, and c
// are colinear (on the same line) throws
// ColinearPointsException; else constructs
// triangle with vertices a, b, and c ordered
// in clockwise rotation (i.e., visiting its
// vertices in a forward traversal visits them
// in clockwise rotation).
```

下面举例说明顺时针旋转，假设点 $p_0=(0, 0)$, $p_1=(20, 0)$, $p_2=(20, 20)$ ，则三角形 t_1 和 t_2 由下面语句产生：

```
TriangleGeometry t1 = new TriangleGeometry(p0, p1, p2)
TriangleGeometry t2 = new TriangleGeometry(p0, p2, p1)
```

按顺时针旋转产生的结果相同：如果用迭代器的next方法去遍历，访问顶点顺序都是 p_0, p_1, p_2 。

[提示：为了测试三点是否在同一直线上，用其中两个点创建LineGeometry对象，用它的classifyPoint方法判断第三个点的位置。你还需定义ColinearPointsException类来扩展RuntimeException类。】

6.14 一个简单多边形称为凸多边形 (convex)，当且仅当任意两个位于多边形内的点连成的直线段位于多边形内。如图6-6中的前面两个为凸多边形，最后边的则不是。显然凸多边形的内角不大于 180° 。

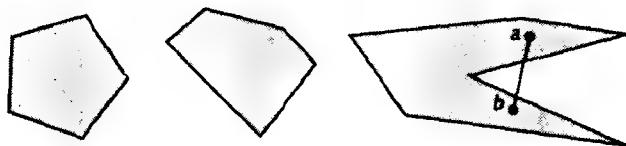


图6-6 凸多边形和非凸多边形

凸多边形 P 的辅助线 (supporting line) 是经过它的某个顶点的一条直线，并且使 P 的全部内点位于这条线的一边。辅助线是与多边形相切的直线。假设 p 是凸多边形 P 外的任意一点，则通过 p 点的辅助线只有两条：右辅助线 (right supporting line) 为通过 p 点位于多边形的右侧；左辅助线 (left supporting line) 为通过 p 点位于多边形的左侧 (参见图6-7)。经过辅助线的凸多边形的点称为切点。

图6-7同时显示了寻找通过点 p 的右辅助线的过程：沿顺时针方法遍历多边形的边，直到找到边 e ，点 p 不在边 e 的左侧 (可以把边 e 无限延长，点 p 位于边上或右侧)；然后从 e 开始继续遍历，直到找到边 f ，如果点 p 位于边 f 的右侧，则边 f 的起始

点（在图6-7中起始点标记为 q ）和 p 的连线为右辅助线。

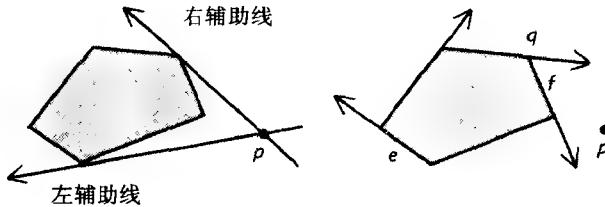


图6-7 左图：通过 p 的辅助线；右图：经过的顶点 q 辅助线的定位过程

下面的过程rightSupportingLine将利用这个策略来找通过点 p 的凸多边形的右辅助线。我们没有把多边形对象作为参数，而是传入它的迭代器iter。此过程有下面三个前置条件：

- 多边形是至少有三个顶点的凸多边形，且这些顶点不在一条直线上。
- 点 p 位于多边形的外部。
- 多边形的顶点和边要顺时针旋转（使用前面练习的TriangleGeometry类产生的三角形将满足这一点）。

此过程有下面两个后置条件：

- 通过迭代器iter找出通过点 p 的右辅助线的顶点。
- 该过程返回一个LineGeometry对象表示这条辅助线。

下面是过程的定义：

```
public static LineGeometry
rightSupportingLine(PolygonIterator iter,
                     PointGeometry p)
    throws NullPointerException {
    // REQUIRES: iter's underlying polygon is convex,
    // has at least 3 noncolinear vertices, has
    // clockwise rotation, and does not contain p.
    // MODIFIES: iter
    // EFFECTS: If iter or p is null throws
    // NullPointerException; else iter is made to
    // point to some vertex through which passes
    // the right supporting line through point p, and
    // this supporting line is returned.
    boolean pNotLeft = false;
    PointGeometry a = iter.point();
    while (true) {
        iter.next();
        PointGeometry b = iter.point();
        // line is the current edge of the polygon
        LineGeometry line = new LineGeometry(a, b);
        int classification = line.classifyPoint(p);
        if (pNotLeft &&
            (classification == LineGeometry.LEFT)) {
            // support point is found; set iter and
            // return the supporting line
            iter.prev();
            return new LineGeometry(p, iter.point());
        } else if (classification != LineGeometry.LEFT)
            // found some edge that p does not lie
            // to the left of
            pNotLeft = true;
    }
}
```

```

    // advance to the next edge
    a = b;
}
}

```

注意布尔变量pNotLeft只有在找到边 e 后才置成true，且只有找到边 e 后再继续才可以找到右辅助线的顶点。

在这个练习中，过程rightSupportingLine为类DynamicPolygons中的静态方法。按照类似的方法，定义一个leftSupportingLine过程来寻找左辅助线：

```

public static LineGeometry
leftSupportingLine(PolygonIterator iter,
                   PointGeometry p)
throws NullPointerException {
    // REQUIRES: iter's underlying polygon is convex,
    // has at least 3 noncolinear vertices, has
    // clockwise rotation, and does not contain p.
    // MODIFIES: iter
    // EFFECTS: If iter or p is null throws
    // NullPointerException; else iter is made to
    // point to some vertex through which passes
    // the left supporting line through p, and
    // this supporting line is returned.
}

```

6.15 编写一个图形交互程序playSupportingLines来测试在前面练习中编写的两个过程。程序有两个参数，由它们决定一个规则多边形 P ：

```
> java PlaySupportingLines nbrSides radius
```

由于每个规则多边形都是凸多边形，所以这里的 P 是凸多边形。该程序反复提示用户输入命令：

- point x y 如果点 (x, y) 位于 P 内，程序显示 P 并给出信息“ (x, y) lies inside the polygon”；如果点 (x, y) 位于 P 外，程序显示 P 和点 (x, y) 并画出通过 (x, y) 的两条辅助线。
- quit 退出程序。

你可以用不同的颜色区分左右辅助线，如右辅助线为红色，左辅助线为绿色。

6.16 一个点集 S 的凸壳（convex hull）是指包括 S 中所有点的最小的凸多边形 P 。这里最小的意思是如果另有一个凸多边形 P' 也包含 S 中所有点，则 P' 一定包含 P 。我们用CH(S)表示点集 S 的凸壳（参见图6-8）。

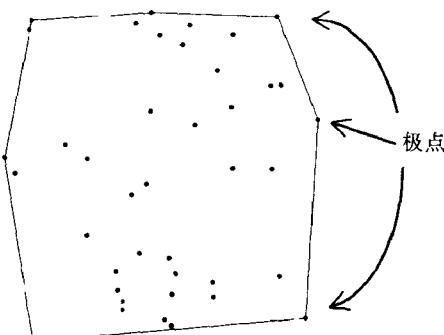


图6-8 一个点集的凸壳

一个有限点集 S 的凸壳可以这样形容：想象一块平木板，在它的上面钉了一些钉子。每个钉子代表点集中的一点，现在用一根橡皮绳拉紧并使它能围住所有的钉子，然后放松它使它以一些钉子为支点形成一个凸多边形形状。需注意的是如果 S 中的点为凸壳的顶点（称为极点（extreme point）），则这个点的内角必然小于 180° 。除了极点外， S 中的其他点或位于凸壳的边上或位于内部。虽然这种方法可以在实际中确定一个点集的凸壳，但它不能用在计算机中。

有很多有趣的算法可以用来创建一个有限点集 S 的凸壳。在这个练习中，我们将用一种称为渐增法的方法。假设 S 包含点 p_0, p_1, \dots, p_{n-1} ， $n \geq 3$ （ S 可以用数组或矢量实现）； $CH(i)$ 表示 S 中前面 i 个点的凸壳，也即 $CH(i) = CH(\{p_0, p_1, \dots, p_{i-1}\})$ 。我们的目标是得到 $CH(n)$ ，也就是 $CH(S)$ 。下面是这个算法的描述：

```
convexHull(S={p0,p1,...,pn-1}) {
    H ← CH(3); // build the convex hull over {p0,p1,p2}
    for (int i = 3; i < n; i++)
        insertPointIntoHull(H, pi); // H ← CH(H ∪ pi)
    return H; // H is now CH(n)=CH(S)
}
```

这个算法的思路是：先用 S 中前面三个点（假设三点不在一直线上）建立凸壳 H ，然后一次一个点把它们逐渐插入到 H 中，在最后 H 等于所需要的 $CH(S)$ 。

把一个点 p 插入到当前凸壳 H 中，用下面的过程实现：

```
static void
insertPointIntoHull(DynamicPolygonGeometry H,
    PointGeometry p)
throws NullPointerException
// REQUIRES: H is a convex polygon with clockwise
// rotation and at least three vertices.
// MODIFIES: H
// EFFECTS: If H or p is null throws
// NullPointerException; else makes H equal
// to CH(H ∪ p), the convex hull of H and p.
```

计算 $CH(H \cup p)$ 是按下列两种情况进行的（ H 是凸多边形， p 是一个点）：

- 1) H 包括点 p 时 显然 $CH(H \cup p) = H$ ，没有其他操作。
- 2) H 不包括点 p 时 找出通过点 p 的 H 的两条辅助线，两条辅助线上的切点把 H 的顶点分成两个链，靠近 p 点的部分称为近链（near chain），另一部分为远链（far chain）（见图 6-9）。用 p 点代替近链，就得到 $CH(H \cup p)$ 。

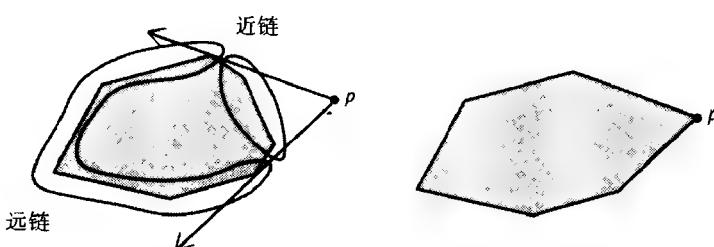


图 6-9 通过 p 的两条辅助线把多边形的边分成两部分

下面的过程 `replaceInnerChain` 实现第二种情况，参数 H 是凸多边形而 p 是 H

外的点，通过修改输入的 H 来产生凸壳 $CH(H \cup p)$:

```
protected static void
replaceInnerChain(DynamicPolygonGeometry H,
    PointGeometry p)
    throws NullPointerException {
    // find left and right supporting lines
    PolygonIterator rightIter = H.iterator();
    rightSupportingLine(rightIter, p);
    PointGeometry rTangentPoint =
        rightIter.point();
    PolygonIterator leftIter = poly.iterator();
    leftSupportingLine(leftIter, p);
    // remove inner chain
    leftIter.prev();
    while (!leftIter.point().equals(rTangentPoint))
        leftIter.remove();
    // insert point p
    leftIter.insertAfter(p);
}
```

至此我们有了创建凸壳的工具，实现静态方法`convexHull`、`insertPointIntoHull`、`replaceInnerChain`，并把它们加入到`DynamicPolygons`类中。下面是`convexHull`方法的说明：

```
static DynamicPolygonGeometry
convexHull(PointGeometry[] points)
    throws NullPointerException,
        IllegalArgumentException,
        ColinearPointsException
// EFFECTS: If points is null or some points[i]
// is null throws NullPointerException; else
// if points.length < 3 throws
// IllegalArgumentException; else if the
// first three points are colinear throws
// ColinearPointsException; else returns
// the convex hull of points.
```

在程序中使用你的`convexHull`过程

```
> java PaintConvexHull n
```

该过程产生 n 个随机点，构造 n 个随机点的凸壳，并输出随机点和它的凸壳。图6-8是当 n 为40时程序显示的结果。

6.17 定义一个过程，它把一个三角形 t 用它包含的某个点 p （在三角形内或三角形上）分裂成一些小的三角形。有两种情况需要考虑：

1) 点 p 位于三角形 t 内（图6-10的第一个图）。 t 被分裂成三个小三角形，分别为 Δabp 、 Δbcp 和 Δcap 。

2) 点 p 位于三角形 t 的某条边上（图6-10的第二个图）。 t 被分裂成两个小三角形，分别为 Δcap 和 Δbcp 。

如果 p 和三角形 t 的某个顶点重合，则不必分裂三角形。下面的过程`splitTriangle`的实现中不考虑这种情况：

```
public static TriangleGeometry[]
splitTriangle(TriangleGeometry t, PointGeometry p)
// REQUIRES: t is a triangle in clockwise
```

```

// rotation, t contains point p, and p does
// not coincide with any vertex of t.
// EFFECTS: Returns an array of triangle
// geometries that result from splitting t
// by point p.
TriangleGeometry[] pieces;
PolygonIterator iter = t.iterator();
// case where p falls along some edge of t
PointGeometry a = iter.point();
for (int i = 0; i < 3; i++) {
    iter.next();
    PointGeometry b = iter.point();
    LineGeometry line = new LineGeometry(a, b);
    if (line.classifyPoint(p)===LineGeometry.ON) {
        iter.next();
        PointGeometry c = iter.point();
        pieces = new TriangleGeometry[2];
        pieces[0] = new TriangleGeometry(a, p, c);
        pieces[1] = new TriangleGeometry(p, b, c);
        return pieces;
    }
    a = b;
}
// case where p falls in the interior of t
a = iter.point();
iter.next();
PointGeometry b = iter.point();
iter.next();
PointGeometry c = iter.point();
pieces = new TriangleGeometry[3];
pieces[0] = new TriangleGeometry(a, b, p);
pieces[1] = new TriangleGeometry(b, c, p);
pieces[2] = new TriangleGeometry(c, a, p);
return pieces;
}

```

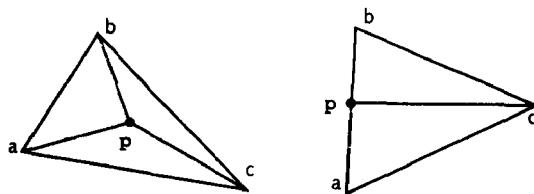


图6-10 分裂三角形的两种情况

把splitTriangle加到类DynamicPolygons中，然后写一个测试图形程序PaintSplitTriangle，它带有8个整型参数，分别表示4个不同的点：

```
> java PaintSplitTriangle x0 y0 x1 y1 x2 y2 x3 y3
```

设点 (x_3, y_3) 包括在 (x_0, y_0) 、 (x_1, y_1) 、 (x_2, y_2) 组成的三角形中。程序应该把分裂出的小三角形用不同的颜色表示。使用splitTriangle过程分裂三角形 t 。

6.18 有限点集 S 的三角形剖分 (triangulation) 是这样一组三角形的集合：它们的顶点都是 S 中的点，并且它们合并起来正好和 S 的凸壳相等。图6-11中就是两个三角形剖分，它们分别由12个点和24个点组成。在这个练习中，我们要设计一个过程 triangulation，用有限点集形成三角形剖分。这里使用的算法是渐增法的一个

变种（参见练习6.16），先用三个点创建一个凸壳，然后逐渐增加点并增大凸壳。每加一个点，就重新计算当前形成的三角形剖分，并保存在一个集合中。`triangulation`返回一个包含组成三角形剖分的三角形的矢量，输入参数是点组成的数组。下面是主过程：

```

public static
Vector triangulation(PointGeometry[] pts)
    throws NullPointerException,
           IllegalArgumentException,
           ColinearPointsException {
    // EFFECTS: If pts is null or pts[i] is null for
    // some i throws NullPointerException; else if
    // pts.length<3 throws IllegalArgumentException;
    // else if the first three points are colinear
    // throws ColinearPointsException; else returns
    // a vector of triangles representing
    // a triangulation of pts.
    if (pts.length < 3)
        throw new IllegalArgumentException();

    // container to hold triangles
    Vector triangles = new Vector();

    // construct CH(3) and add first triangle to
    // the triangulation
    DynamicPolygonGeometry H =
        new TriangleGeometry(pts[0], pts[1], pts[2]);
    triangles.add(
        new TriangleGeometry(pts[0], pts[1], pts[2]));

    // for each i=4,...,pts.length, construct CH(i)
    // and maintain current triangulation
    for (int i = 3; i < pts.length; i++)
        addNewTriangles(H, pts[i], triangles);
    return triangles;
}

```

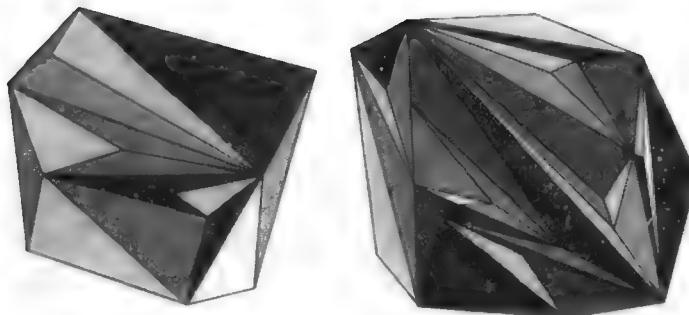


图6-11 左：12个点形成的三角形剖分；右：24个点形成的三角形剖分

过程addNewTriangles

```

addNewTriangles(DynamicPolygonGeometry H,
                PointGeometry p,
                Vector triangles)

```

用三个参数调用，当前凸壳H，保存在矢量`triangles`中的凸壳H的三角形集和将要插入的点p。在增加点p的过程中可能要对矢量`triangles`进行增加或删除三角

形的操作。有两种情况要考虑：(1) H 包含点 p ；(2) H 不包含点 p ；在下面过程的实现中，你可以看到这两种情况的不同处理：

```
public static void
addNewTriangles(DynamicPolygonGeometry hull,
                PointGeometry p,
                Vector triangles) {
    // REQUIRES: triangles is a triangulation of hull.
    // MODIFIES: triangles
    // EFFECTS: Revises triangles to account for the
    //           insertion of point p.
    if (hull.contains(p)) { // case 1
        TriangleGeometry[] affectedTris =
            findAffectedTriangles(p, triangles);
        for (int i=0; i<affectedTris.length; i++) {
            TriangleGeometry t = affectedTris[i];
            TriangleGeometry[] pieces =
                splitTriangle(t, p);
            triangles.remove(t);
            for (int j = 0; j < pieces.length; j++)
                triangles.add(pieces[j]);
        }
    } else // case 2
        splitFan(hull, p, triangles);
}
```

本练习的下面部分主要关注两种情况。

(1) 点 p 包含在当前三角形剖分对应的凸壳中。所有受点 p 影响的三角形可以调用 `findAffectedTriangles` 过程获得，然后把每个这样的三角形从三角形剖分中删除，然后以由它们和点 p 产生的小三角形替换。

过程 `findAffectedTriangles` 用来查找受影响的三角形，它返回一个三角形数组，这些三角形需要分成小的三角形。该过程测试点 p 和每个三角形的关系，下面列出所有情况：

- p 和某个三角形的顶点重合。返回长度为0的数组（没有三角形需要分割）。
- p 只位于某个三角形的边上。返回包含对应三角形的数组。这种情况， p 位于凸壳的边上。
- p 位于两个三角形公有的边上。返回包含这两个三角形的数组。
- p 位于某一个三角形的内部。返回包含对应三角形的数组。

下面是 `findAffectedTriangles` 过程的实现：

```
protected static TriangleGeometry[]
findAffectedTriangles(PointGeometry p,
                     Vector triangles) {
    // REQUIRES: p and triangles are not null, and
    //           p lies in at least one triangle.
    // EFFECTS: Returns an array of those triangles
    //           that contain p and must be split into either
    //           two or three pieces (as in Figure 6.10).
    TriangleGeometry[] affectedTris =
        new TriangleGeometry[2];
    int nbrAffectedTris = 0;
    nexttriangle:
    for (int i = 0; i < triangles.size(); i++) {
        TriangleGeometry t =
```

```

        (TriangleGeometry)triangles.get(i);
    if (t.contains(p)) {
        PolygonIterator iter = t.iterator();
        // if p coincides with a vertex of t,
        // then no triangles are affected by p
        for (int j = 0; j < 3; j++, iter.next())
            if (p.equals(iter.point()))
                return new TriangleGeometry[0];
        // if p lies along an edge of t, keep t
        PointGeometry a = iter.point();
        for (int j = 0; j < 3; j++) {
            iter.next();
            PointGeometry b = iter.point();
            LineGeometry line = new LineGeometry(a, b);
            if (line.classifyPoint(p)==LineGeometry.ON){
                affectedTris[nbrAffectedTris++] = t;
                if (nbrAffectedTris == 2)
                    return affectedTris;
                else continue nexttriangle;
            }
            a = b;
        }
        // p lies in the interior of t,
        // so keep t and return
        TriangleGeometry[] resTris =
            new TriangleGeometry[1];
        resTris[0] = t;
        return resTris;
    }
    // end for loop
    // p lies on one triangle's edge along
    // the convex hull boundary
    if (nbrAffectedTris != 1)
        throw new IllegalStateException();
    TriangleGeometry[] resTris =
        new TriangleGeometry[1];
    resTris[0] = affectedTris[0];
    return resTris;
}

```

(2) 点 p 不包含在当前三角形剖分对应的凸壳中。过程addNewTriangles产生一组新的三角形, 形似扇形(见图6-12)。过程调用

```
splitFan(H, p, triangles);
```

将用 H 和 p 点产生新的 $H = CH$ ($H \cup p$)。另外, 把点 p 和近链上的点组成新的三角形增加到矢量triangles中。下面是这一过程的实现:

```

public static
void splitFan(DynamicPolygonGeometry H,
              PointGeometry p,
              Vector triangles) {
    // REQUIRES: H is convex, contains at least
    // three vertices, and does not contain p.
    // MODIFIES: H, triangles
    // EFFECTS: Adds to triangles the fan of
    // triangles between p and H, and changes H
    // to CH(HUp).
    // find left and right supporting lines
    PolygonIterator rightIter = H.iterator();

```

```

rightSupportingLine(rightIter, p);
PointGeometry rTangentPoint = rightIter.point();
PolygonIterator leftIter = H.iterator();
leftSupportingLine(leftIter, p);
// build triangles and remove inner chain
PointGeometry a = leftIter.point();
leftIter.prev();
while (!leftIter.point().equals(rTangentPoint)) {
    PointGeometry b = leftIter.point();
    triangles.add(new TriangleGeometry(a, b, p));
    a = b;
    leftIter.remove();
}
triangles.add(
    new TriangleGeometry(a, leftIter.point(), p));
// insert point p
leftIter.insertAfter(p);
}

```

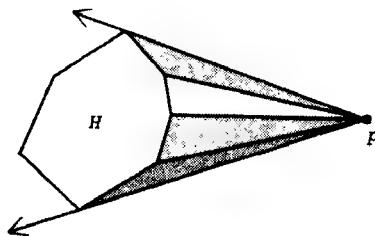


图6-12 凸多边形H和p之间的扇形三角形

把这个练习中增加的过程加入到类DynamicPolygons中，然后写一个图形程序名为PaintTriangulation，它产生n个随机点，用这些随机点产生一个三角形剖分，并把它们画出来。程序执行方法为：

```
> java PaintTriangulation n
```

其中n为所产生的随机点数。图6-11就是使用这个程序产生的两个图形。

6.2.4 迭代器模式的结构和应用

迭代器模式中定义了一个称为迭代器的对象，使用它可遍历访问一个聚集中的各个元素，而又不暴露它的内部结构。迭代器提供了各种各样的访问操作和记录当前遍历位置的方法。例如，一个链表的迭代器可按顺序访问链表中的元素，并可以返回最近访问元素的位置；与此相似，一个二叉树迭代器可以按左子节点、右子节点和父节点顺序访问其中元素，同时记录最近一个访问元素的位置。一般地说，迭代器对聚集的访问顺序是与聚集本身的结构和客户的要求有关的。

迭代器模式负责访问和遍历聚集对象并将其放到迭代器对象中，这使迭代器具有以下几个优点：

第一，遍历顺序可以随意设定。如在本章中定义的多边形迭代器是沿着多边形的边访问其中的顶点，当然可以改成其他的访问顺序，如按它们在平面上的位置从左到右，或按它们到某个点的位置的远近。这对复杂的聚集（如多边形）是很重要的，因为它们可能需要不同的访问顺序。另外迭代器实现不同的访问顺序策略也常常用来满足客户的不

同需求。

第二, 简化了聚集的接口。由于聚集不必按客户要求的每一种方式提供对元素的访问, 聚集的接口因此而大大简化; 又由于迭代器支持控制抽象, 客户的工作也因此简化。只需对迭代器的控制抽象进行操作, 就足以完成对聚集对象的各种各样的遍历。

第三, 不同的遍历可以同时对同一个聚集对象进行操作, 每种遍历都由一个迭代器对象来完成。在练习6.16的replaceInnerChain过程中就利用了这种特征。

图6-13中的类图表达了迭代器模式的结构, 图中的五个组成部分如下:

- **Iterator** 接口 描述访问某一类聚集元素的操作。
- **ConcreteIterator**类 实现Iterator接口功能, 并且每个ConcreteIterator对象都要记录所访问聚集中的当前元素位置。
- **Aggregate** 接口 描述聚集的操作, 包括创建迭代器的操作。
- **ConcreteAggregate**类 实现Aggregate接口。
- **Client** 通过Iterator接口与ConcreteAggregate交互。

图6-13中UML聚合关系中*表示可以有多个ConcreteIterator对象可以共享同一个ConcreteAggregate对象, 这也正是迭代器设计模式的第三个特点。

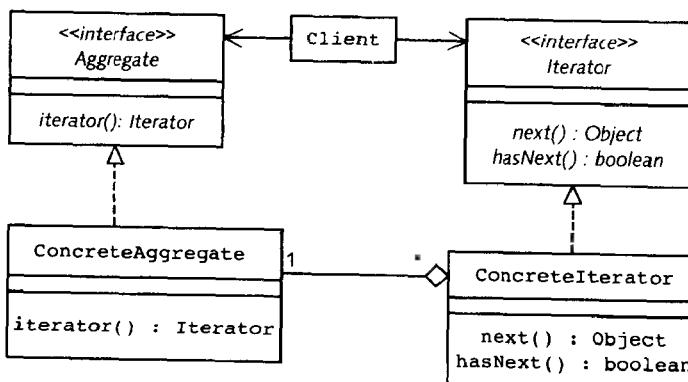


图6-13 迭代器模式的结构

图6-13中的类比以前见到的要详细些(具体解释请见附录C)。每个类由三个部分组成: 最上部分为它的类名; 中间部分为相关域和它们的类型(这部分可以为空); 最后一部分为类的相应操作和它们的返回类型。Iterator接口定义两个操作next和hasNext, 用斜体字表示它们为Iterator接口的抽象操作。next返回一个Object类型, 而hasNext返回boolean类型。这两个方法的实现在它的子类ConcreteIterator中完成。同样Aggregate接口定义了iterator抽象方法, 该方法返回一个聚集迭代器, 由它的子类ConcreteAggregate实现这个方法。

本节中的多边形迭代器的例子要比图6-13描述的简单, 具体来说PolygonIterator接口对应于图6-13中的Iterator, DynamicPolygonIterator类对应于ConcreteIterator, 而DynamicPolygonGeometry提供了Aggregate抽象接口又提供了ConcreteAggregate实现。需要注意的是Iterator中所概括的方法next和hasNext在PolygonIterator中并没有同样名字的方法, 这两个方法是概括了Iterator所要实现的取得当前元素、定位到下一个元素和判断是否到达边界(最后一个

元素) 的功能, 显然这些功能可以用其他方法和域来实现。

6.3 模板方法设计模式

模板方法模式用于定义一个算法。一个算法的某些步骤和另一个算法的对应部分可能会有很大变化。负责一步一步实现算法的方法称为模板方法 (template method), 由于计算的步骤的变化, 这些变化的步骤常常是由其他子类实现, 就是说模板方法会调用一个或多个抽象方法, 这些抽象方法是由不同的子类实现的。在本节中我们将使用这种模式设计类来表示布尔几何图形。

6.3.1 布尔几何图形

到目前为止, 我们所涉及的区域几何图形都是简单图形: 矩形、椭圆、多边形。事实上有很多比这些图形更复杂的图形, 但复杂图形常常可以由这些简单图形拼组成。图6-14就是组合图形的例子, 图中的阴影图形就是通过组合区域图形经过三种特殊的运算组合出来的, 称为布尔几何图形。常用的三种特殊运算为: 并 (union)、交 (intersection)、差 (difference)。这些运算称为布尔形状运算 (boolean shape operation)。图6-14中, 第一个半月图是由两个圆的交叉部分组成; 第二个图形是在一个多边形中挖去一个圆组成; 第三个梨状图形由两部分组成: 两个叶子是两个半月形, 它的身子是由一个圆和一个椭圆合并而成。

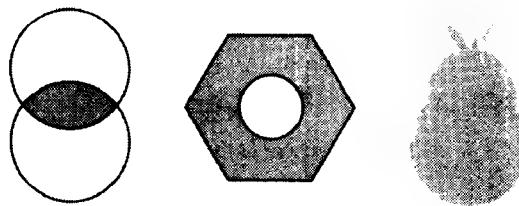


图6-14 布尔几何图形

Java 2D API中图形组合操作是由Area类支持的, Area类实现了Shape接口, 它支持封闭区域形状图形的布尔运算。下面给出的Area的类框架说明了几个我们需要的运算:

```
public class java.awt.geom.Area
    implements java.awt.Shape {
    public Area(Shape s)
        // REQUIRES: s is not null.
        // EFFECTS: Initializes this area from the shape s.

    public boolean contains(double x, double y)
        // EFFECTS: Returns true if this area contains
        // the point (x,y); else returns false.

    public void add(Area a)
        // REQUIRES: a is not null.
        // MODIFIES: this
        // EFFECTS: Sets the shape of this area to
        // the union of this and a
        // (i.e., this_post = this  $\cup$  a).

    public void intersect(Area a)
        // REQUIRES: a is not null.
```

```

// MODIFIES: this
// EFFECTS: Sets the shape of this area to
//   the intersection of this and a
//   (i.e., this_post = this ∩ a).

public void subtract(Area a)
// REQUIRES: a is not null.
// MODIFIES: this
// EFFECTS: Sets the shape of this area to
//   the difference of this and a
//   (i.e., this_post = this - a).
}

```

类Area的布尔图形运算add、intersect和subtract是增变器，它们修改对象的状态。如果a1和a2都是Area对象，则过程调用

```
a1.intersect(a2);
```

修改a1的状态，即a1的形状变成a1和a2的交叉部分，而a2的状态不受这一操作的影响。

同样，我们可以用Area对象构造两个AreaGeometry对象a和b的交叉图形，先取出对象a和b的形状，把它们转成Area对象，然后得到它们的交叉图形，这种策略用在下面的过程中，此过程返回一个表示两个区域几何图形的交叉部分：

```

static Shape shapeOfIntersection(AreaGeometry a,
                                  AreaGeometry b) {
    Area aArea = new Area(a.shape());
    Area bArea = new Area(b.shape());
    aArea.intersect(bArea);
    return aArea;
}

```

例如，下面的代码段将产生图6-14中第一个图形的效果：

```

AreaGeometry topCircle =
    new EllipseGeometry(20, 20, 10, 10);
AreaGeometry bottomCircle =
    new EllipseGeometry(20, 26, 10, 10);
Shape lune = shapeOfIntersection(topCircle, bottomCircle);

```

有趣的是，过程shapeOfIntersection的思路同样可以用在区域几何图形的并和差中，只需修改它的最后一步即可。假定布尔形状运算是由一个称为applyOp的过程完成的，它有两个Area对象参数，布尔组合结果返回到第一个参数中。applyOp完成的是哪种布尔运算是由它的实现决定的。下面的程序段返回两个参数的并、交还是差是由applyOp实现的是三种布尔形状运算的哪一种决定的：

```

public Shape shape(AreaGeometry a, AreaGeometry b) {
    Area aArea = new Area(a.shape());
    Area bArea = new Area(b.shape());
    applyOp(aArea, bArea);
    return aArea;
}

```

上面所定义的过程shape就是一个典型的模板方法。它定义了一个算法，但其中的一些步骤没有实现，也就是applyOp指定的运算没有实现。大家会问：applyOp在哪儿实现呢？答案是在它的子类中。模板方法模式的思想是在抽象父类中放置一个模板方法，模板方法调用一个抽象方法，抽象方法最后在子类中实现定义。此外，父类定义必要的抽象方法以使它的子类实现它们。在这种设计模式中，这些抽象方法也称为钩子方法

(hook method).

下面看一看模板设计模式在布尔几何图形问题中如何应用。如图6-15所示，抽象类 BooleanGeometry 中包含了前面定义过的 shape 过程，也包括抽象过程 applyOp，applyOp 的具体实现是在 BooleanGeometry 的子类中，每一个对应一种布尔形状运算。这里 shape 是模板方法， applyOp 是钩子方法。

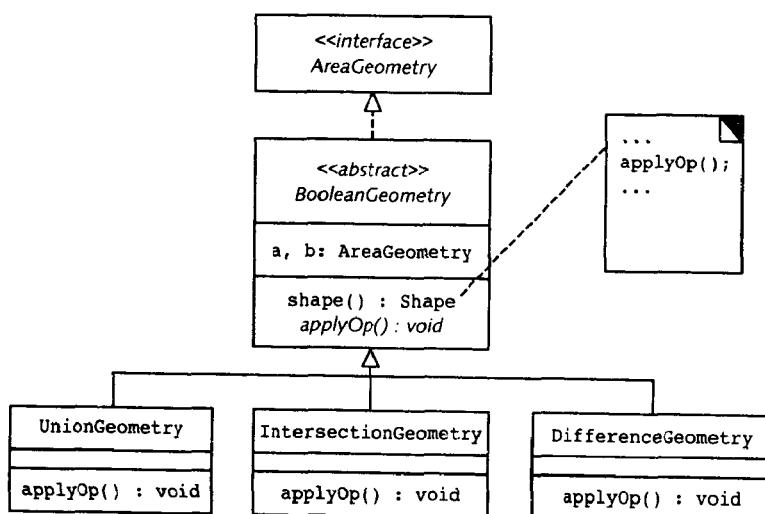


图6-15 使用模板方法的图形组合类

`BooleanGeometry`类中有两个域`a`、`b`，它们都是`AreaGeometry`对象。每个类方框的第三个格是相关操作和它们的返回值类型。显然`BooleanGeometry`定义了两个相关操作：`shape`和`applyOp`。`applyOp`用斜体表示它是`BooleanGeometry`类的抽象操作，三个子类中都继承了其父类`BooleanGeometry`的域并实现了`applyOp`操作。`BooleanGeometry`中`shape`方法虚线连接的注释表明`shape`的实现形式并强调它调用了抽象方法`applyOp`。这也表明模板设计模式的特点：模板方法调用一个或多个钩子方法，钩子方法在模板方法的类中有抽象定义但没有具体实现。

下面来实现 BooleanGeometry 类。因为布尔几何图形封闭了一个区域，所以该类要实现 AreaGeometry 接口。这就意味着 BooleanGeometry (或它的子类) 要实现下列几个方法： contains、shape、translate。下面是该类的类定义：

```
public abstract class BooleanGeometry
    implements AreaGeometry {

    protected AreaGeometry a, b;

    protected BooleanGeometry(AreaGeometry a, AreaGeometry b)
        throws NullPointerException {
        // EFFECTS: If a or b is null throws
        //   NullPointerException; else constructs
        //   the Boolean combination of a and b.
        if ((a==null) || (b==null))
            throw new NullPointerException();
        this.a = a;
        this.b = b;
    }
}
```

```

}

public boolean contains(int x, int y) {
    return shape().contains(x, y);
}

public boolean contains(PointGeometry p)
    throws NullPointerException {
    return contains(p.getX(), p.getY());
}

public void translate(int dx, int dy) {
    a.translate(dx, dy);
    b.translate(dx, dy);
}

// the template method shape expects the applyOp method to
// be implemented by concrete descendants of this class.
public Shape shape() {
    Area aArea = new Area(a.shape());
    Area bArea = new Area(b.shape());
    applyOp(aArea, bArea);
    return aArea;
}

// hook method required by the shape method.
protected abstract void applyOp(Area aArea, Area bArea);
// REQUIRES: aArea and bArea are not null.
// MODIFIES: aArea
// EFFECTS: Sets aArea to the Boolean combination
// of aArea and bArea.
}

```

两个参数的 `contains` 方法将它的工作委托给布尔几何图形的形状，这和类 `PolygonGeometry` 中的 `contains` 是相似的。注意，`translate` 方法通过平移两个部分的每一部分来平移布尔几何图形。

子类实现由其父类 `BooleanGeometry` 声明的 `applyOp` 方法，下面是表示两个区域几何图形交的类 `IntersectionGeometry` 的类定义：

```

public class IntersectionGeometry extends BooleanGeometry {

    public IntersectionGeometry(AreaGeometry a, AreaGeometry b)
        throws NullPointerException {
        super(a, b);
    }

    protected void applyOp(Area aArea, Area bArea) {
        aArea.intersect(bArea);
    }
}

```

练习

6.19 试实现 `UnionGeometry` 类和 `DifferenceGeometry` 类。

6.3.2 半月图

半月图 (lune) 是由两个圆相交而成。本节中将设计一个画半月图的程序，并封装成

一个半月图类。

1. 画半月图

在设计程序前，先假设组成半月图的两个圆半径相同，它们的圆心在y轴上且离坐标原点距离相同。图6-16（左）标出了决定半月图形状和大小的两个参数：半径指每个圆的半径；偏移量指坐标原点到每个圆心的距离。

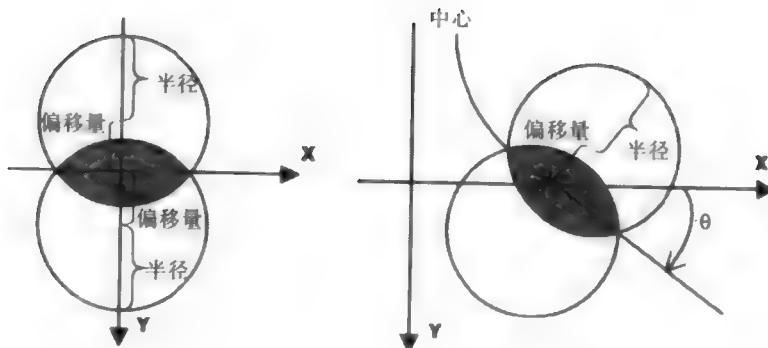


图6-16 描述半月图的参数

程序 `PaintLune` 使用 `MyGraphicsProgram` 模板，它要求定义三个方法 `parseArgs`、`MakeContent`、`paintComponent`，分别用来分析输入参数、创建绘图环境、画出图形内容。`PaintLune` 有两个参数 `radius` 和 `offset`，分别指定半月图的半径和偏移量，执行方法如下：

```
> java PaintLune offset radius
```

下面是 `parseArgs` 方法的实现：

```
// static fields of PaintLune class
protected static int offset, radius, diam;

// static method of PaintLune class
public static void parseArgs(String[] args) {
    if (args.length != 2) {
        String s="USAGE: java PaintLune offset radius";
        System.out.println(s);
        System.exit(1);
    }
    offset = Integer.parseInt(args[0]);
    radius = Integer.parseInt(args[1]);
    diam = 2 * radius;
}
```

半月图被存储在 `Figure` 类型的 `luneFigure` 域中，并将用草绿色填充。`makeContent` 方法定义如下：

```
// field of PaintLune class
protected Figure luneFigure;

// method of PaintLune class
public void makeContent() {
    PointGeometry p =
        new PointGeometry(-radius, -radius-offset);
    AreaGeometry topCircle =
```

```

        new EllipseGeometry(p, diam, diam);
p.translate(0, 2*offset);
AreaGeometry botCircle =
    new EllipseGeometry(p, diam, diam);
BooleanGeometry lune =
    new IntersectionGeometry(topCircle, botCircle);
Painter painter =
    new FillPainter(new Color(33,140,33));
luneFigure =
    new Figure(lune, painter);
}

```

下面是paintComponent方法的实现，其中最后三行程序是将绘图环境的坐标系移到屏幕中心然后输出半月图：

```

// method of PaintLune class
public void paintComponent(Graphics g) {
    super.paintComponent(g);
    Graphics2D g2 = (Graphics2D)g;
    g2.setRenderingHint(RenderingHints.KEY_ANTIALIASING,
        RenderingHints.VALUE_ANTIALIAS_ON);
    Dimension d = getFrame().getContentSize();
    g2.translate((int)(d.width/2), (int)(d.height/2));
    luneFigure.paint(g2);
}

```

练习

6.20 实现本节中描述的PaintLune程序。

2. 半月图类

下面将定义一个类，把半月图封装起来，并要一般化半月图的位置和方向。除了前面PaintLune程序中用的offset和radius参数外，还需要增加两个参数：center——半月图的中心点；theta——半月图的主轴和x轴的正向夹角（见图6-16右图），以度为单位。LuneGeometry类提供了一个以这些参数为值的构造器如下：

```

public LuneGeometry(PointGeometry center, int offset,
    int radius, double theta)

```

LuneGeometry类定义了AreaGeometry类型的域lune存储代表半月图的布尔几何图形，并由方法ComputeLune依据其他变量计算设置lune的值：

```

// field of LuneGeometry class
protected AreaGeometry lune

```

由于半月图包含有区域的概念，所以LuneGeometry类实现了AreaGeometry接口。下面是该类的完整定义：

```

public class LuneGeometry implements AreaGeometry {

    protected AreaGeometry lune;
    protected PointGeometry center;
    protected int offset, radius;
    protected double theta;

    public LuneGeometry(PointGeometry center,
        int offset, int radius, double theta)
        throws NullPointerException {

```

```

// EFFECTS: If center is null throws
//   NullPointerException; else constructs a
//   lune based on the supplied parameters.
this.center = new PointGeometry(center);
this.offset = offset;
this.radius = radius;
this.theta = theta;
computeLune();
}

public Shape shape() {
    return lune.shape();
}

public void translate(int dx, int dy) {
    center.translate(dx, dy);
    computeLune();
}

public boolean contains(int x, int y) {
    return shape().intersects(x-0.01, y-0.01, .02, .02);
}

public boolean contains(PointGeometry p) {
    return contains(p.getX(), p.getY());
}

protected void computeLune() {
    // REQUIRES: center is not null.
    // MODIFIES: lune
    // EFFECTS: Sets lune to values in the center,
    //   offset, radius, and theta fields
    int diam = 2 * radius;
    TransformablePointGeometry topP =
        new TransformablePointGeometry(center.getX(),
                                       center.getY()-offset);
    topP.rotate(theta, center);
    topP.translate(-radius, -radius);
    AreaGeometry topCircle =
        new EllipseGeometry(topP, diam, diam);
    TransformablePointGeometry botP =
        new TransformablePointGeometry(center.getX(),
                                       center.getY()+offset);
    botP.rotate(theta, center);
    botP.translate(-radius, -radius);
    AreaGeometry botCircle =
        new EllipseGeometry(botP, diam, diam);
    lune =
        new IntersectionGeometry(topCircle, botCircle);
}
}

```

练习

6.21 在LuneGeometry类中哪些部分是没有变化的?
 6.22 描述组成下面半月图的两个圆的位置、宽度和高度:

```
new LuneGeometry(new PointGeometry(10,20),100,120,45)
```

6.23 重写程序PaintLune, 使它带有三个运行参数: *theta*, *x*和*y*:

```
> java PaintLune offset radius [theta [x y]]
```

*x*和*y*表示图形的中心, *theta*为它和*x*轴的夹角。其中*theta*, *x*和*y*省略时值为0。

6.24 设计类NutGeometry来表示图6-14中间的坚果状图, 它实际上是由一个多边形从中间挖掉一个圆形形成的。构造器有几个参数: 环中心点位置center, 多边形边数n, 多边形半径outerRadius, 圆半径innerRadius, 多边形旋转角度twist, 下面是构造器描述:

```
public NutGeometry(PointGeometry center, int n,
    int outerRadius, int innerRadius, double twist)
    throws NullPointerException,
    IllegalArgumentException
// EFFECTS: If center is null throws
// NullPointerException; else if n<=2, or
// innerRadius or outerRadius are <= 0 throws
// IllegalArgumentException; else creates a
// nut with the specified parameters.
```

同LuneGeometry一样, 这个类要实现接口AreaGeometry。你或许可以做成像LuneGeometry类那样的类模板, 使用类RegularPolygonGeometry和EllipseGeometry的布尔差形成一个环。

完成程序PaintNut, 使用前面定义NutGeometry类, 执行它输出一个*n*条边的具有指定半径和旋转角度的中心在点(*x*, *y*)的坚果状图, 调用如下:

```
> java PaintNut n outerRadius innerRadius [theta [x y]]
```

默认的旋转角度为0, 默认的中心是原点。请看一下当innerRadius的值大于等于outerRadius时, 会产生什么样的几何图形?

6.3.3 构造区域几何图形

在上一节中为了构造一个半月图我们组合了两个基本几何图形。像基本几何图形一样, 进行布尔几何图形的并、交、差也是可以的。通过对已存在的几何图形(基本几何图形和布尔几何图形)应用布尔形状运算来构造一个新形状过程称为构造区域几何图形(constructive area geometry, CAG)。

该过程的结果可以描述为一棵几何图形的二叉树, 称为CAG树。如图6-17所示, 右图为一只眼的CAG树, 它由一个半月图去掉一个大圆(表示眼睛的虹膜), 再加上一个小圆(表示眼珠)构成。CAG树的叶节点代表基本图形, 如椭圆、多边形和矩形等; 而每个中间节点标有布尔运算符, 表示它是由两个子节点经过这个运算得到的。

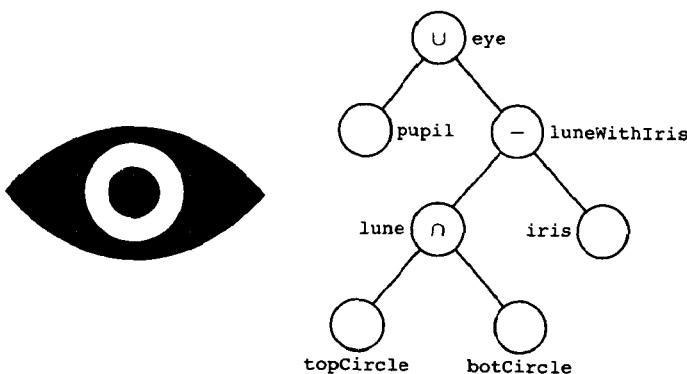


图6-17 使用CAG构造眼睛图

图6-17左边眼睛图可由下面的代码段构造，先定义了几个必须变量，然后构造每个节点，节点名和右图树中一致。该代码段包括几个指定眼睛图的维数的参数：

```

int luneRad,      // lune radius (see Figure 6.16)
luneOffset,      // lune offset (see Figure 6.16)
luneDiam,        // lune diameter: 2*luneRad
irisRad,         // iris radius
irisDiam,        // iris diameter: 2*irisRad
pupilRad,        // pupil radius
pupilDiam;       // pupil diameter: 2*pupilRad

AreaGeometry topCircle =
  new EllipseGeometry(-luneRad, -luneOffset-luneRad,
                      luneDiam, luneDiam);
AreaGeometry botCircle =
  new EllipseGeometry(-luneRad, luneOffset-luneRad,
                      luneDiam, luneDiam);
BooleanGeometry lune =
  new IntersectionGeometry(topCircle, botCircle);
AreaGeometry iris =
  new EllipseGeometry(-irisRad, -irisRad,
                      irisDiam, irisDiam);
BooleanGeometry luneWithIris =
  new DifferenceGeometry(lune, iris);
AreaGeometry pupil =
  new EllipseGeometry(-pupilRad, -pupilRad,
                      pupilDiam, pupilDiam);
BooleanGeometry eye =
  new UnionGeometry(pupil, luneWithIris);

```

练习

6.25 基于模板MyGraphicsProgram编写一个Java应用程序PaintEye创建一个眼睛图（见图6-17），该应用程序以四个参数调用：

```
> java PaintEye luneRadius luneOffset irisRad pupilRad
```

参数定义同前面代码段中使用的参数，其中图6-17中图形由下面参数得到：

```

luneRadius: 160
luneOffset: 120
irisRad: 30
pupilRad: 15

```

6.26 编写交互程序PlayBoolean构造并显示布尔几何图形。程序包含一个当前图形，每一个命令都是对当前图形和一个新的规则多边形（由参数n和radius确定）进行组合，得到下一步的当前图形。命令还指出新的布尔几何图形是由并、交还是差形成的。（当然程序刚开始时当前布尔几何图形为空，第一条命令产生一个规则多边形。）下面为命令解释：

- **union n radius** 合并当前图形和一个带有指定半径且中心在原点（框架的中心）的规则n边形。
- **intersection n radius** 得到当前图形和一个带有指定半径且中心在原点（框架的中心）的规则n边形的交叉部分。
- **difference n radius** 得到当前图形和一个带有指定半径且中心在原点（框架的中心）的规则n边形的不同部分。
- **quit** 退出程序。

图6-18的三个图形分别由下列交互命令产生：

```
> java PlayBoolean
? union 4 100
? difference 6 60      // left figure of Figure 6.18
? union 4 40           // middle figure
? intersection 3 100   // right figure
? quit
```



图6-18 练习6.26程序PlayBoolean产生的图形

6.27 在练习6.26的基础上增加下面两个命令，以便可以指定每个规则多边形的中心和旋转角度：

- **twist *d*** 由union, difference和intersection命令指定的下一个多边形旋转*d*度。
- **center *x* *y*** 由union, difference和intersection命令指定的下一个多边形中心点在(*x*,*y*)。

下面命令将产生图6-19中的三个图形：

```
> java PlayBoolean
? union 4 100
? center 50 0
? union 4 100      // left figure of Figure 6.19
? twist 45
? difference 4 60 // middle figure
? center -25 0
? twist 10
? difference 8 20 // right figure
? quit
```

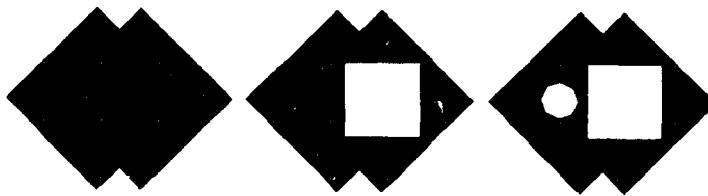


图6-19 练习6.27程序PlayBoolean产生的图形

6.3.4 模板方法模式的结构和应用

模板方法模式用于定义一个算法，这个算法的其中一些步骤是由子类实现的。实现这个算法的方法称为模板方法。模板方法会调用一个或多个抽象方法（钩子方法），这些抽象方法表示可变化步骤，抽象方法是由子类实现。子类以不同的方式完成算法。图6-20为模板方法模式的一般结构，在这个结构图中共有两种类型的元素：

- **AbstractClass**（抽象类） 定义了模板方法templateMethod和模板方法使用的抽象钩子方法op1和op2。

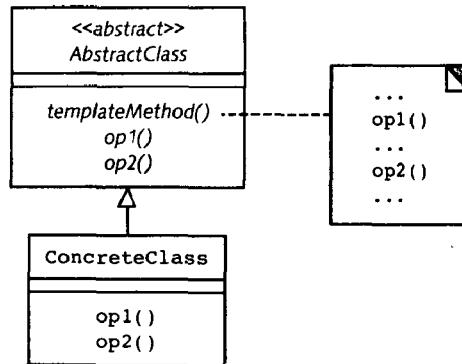


图6-20 模板方法模式的结构

- **ConcreteClass** (具体类) 扩展了 **AbstractClass**, 其中每一个这样的类实现继承的模板方法所要求的抽象钩子方法。

在前面的例子中, 类 **BooleanGeometry** 是 **AbstractClass**, 方法 **shape** 是模板方法, **shape** 依赖于应用到布尔形状运算上的抽象方法 **applyOp**, 反过来它又由三种不同类型的 **Concreteclass** 实现。类 **UnionGeometry**, **IntersectionGeometry** 和 **DifferenceGeometry** 是三个 **ConcreteClass**。

在这种设计模式下, 由模板方法实现的算法是可重用的, 每个 **Concreteclass** 只需实现模板方法中的抽象方法。由于模板方法模式结构清晰, 所以它既可以减少重新创建子类的复杂又可以减少实现算法中的错误, 同时可以确保每个子类实现它要实现的部分。如果一个实现类没有实现父类定义的抽象方法, 那么它是不能实例化的。

6.4 组合设计模式

顾名思义, 组合模式是用一组原子组件 (primitive) 组成一个组合体 (composite)。因为原子组件和组合体都可以用来组成新的组合体, 所以组合体用树状结构表示可以有任意层深。这种设计模式的关键是定义了一个由原子组件和组合体实现的接口, 使客户统一看待原子组件和组合体, 一致看作是组件 (component)。

Java的Abstract Window Toolkit (AWT) 就使用这种模式。用户能用它提供的GUI原子组件组合他希望的任何图形 (在3.4.2中有讨论)。GUI原子组件有按钮、列表、标记、文本域、容器。容器是可以包含其他原子组件的组件, 又因为它是一种组件, 所以它可以包含它自己, 即允许组件和容器任意嵌套。这里的容器和组件的角色是相同的, 因此客户可以发送一个 **repaint** 消息给一个组件, 而不用搞清楚它是原子组件 (如按钮或标记) 还是一个容器。

在本节中, 我们用组合设计模式解决图形组合问题, 用简单图形组合复杂图形。

6.4.1 组合图

到目前为止, 我们已掌握如何设计简单图: 蓝色的矩形、绿色边线的多边形、紫色的布尔几何图形。实际上, 我们所要处理的图形要复杂得多, 如图6-21所示。这种组合图 (composite figure) 是由简单图或组合图组成的, 每个组合图又是由组合图或简单图组合成

的，如此重复，便组成了树状层次图。它的叶子节点是简单图，根节点是最后的组合图。

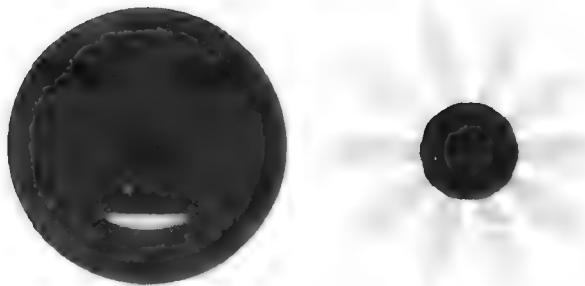


图6-21 两个组合图

使用组合图有许多优点，一是组合图在很多方面可以像简单图一样。例如，你可以定义一个组合图类用来画一张脸或一朵花，在需要的地方直接将这些类的实例输出到绘图环境中。二是可以根据需要修改其中任意一个组件而不影响其他部分。例如修改脸上的眼睛的颜色或形状并不影响脸上的其他部分。三是可以方便地在其上增加或删除组件，如给花增加花茎或从脸上去掉嘴巴。

我们定义一个**GroupNode**类表示组合图，**GroupNode**对象维护一组简单图组件。下面代码使用**GroupNode**绘制图6-21中的脸形图：

```
GroupNode face = new GroupNode();
face.addChild(contour);
face.addChild(leftEye);
face.addChild(rightEye);
face.addChild(nose);
face.addChild(mouth);
```

在上面代码段中，变量**contour**、**lefteye**、**righteye**、**nose**和**mouth**引用类**Figure**实例。例如，**lefteye**和**righteye**都指一个蓝色的半月图，**mouth**是一个绿色的布尔几何图形：由一个圆去掉一个矩形（颜色任由你想象）。图6-22左边的树状层次图更直观地表达了它们之间的关系。

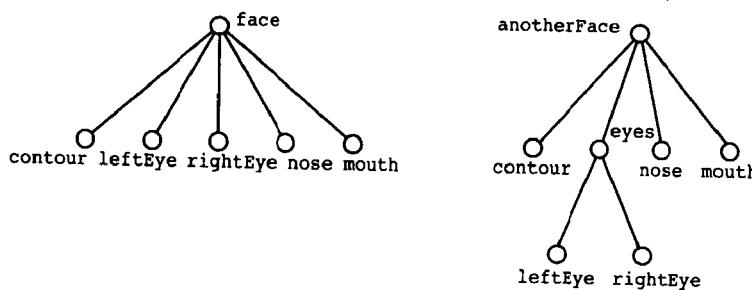


图6-22 图6-21中的脸形图的两种情景图

图6-22的树状层次图称作情景图（scene graph），其中的组成情景图的元素称为节点（node）。左图中有六个节点：**GroupNode**类对象**face**和它的五个**Figure**类对象。

在情景图中，**GroupNode**的组件称为子节点（children）。子节点是按索引顺序排列的，即第一个的索引是0，其他依次增加。子节点的索引顺序很重要，因为它影响画图的顺序。

也就是说，画图时子节点索引值小的先画，索引值大的后画。

那么，像 `face` 这样的 `GroupNode` 应该支持哪些行为呢？首当其冲是要有将一个 `GroupNode` `paint` 到绘图环境 `g2` 的功能：

```
face.paint(g2);
```

其次是使用 `addChild` 方法增加子节点和使用 `removeChild` 方法删除子节点；再次是使用 `child` 方法按索引值访问子节点，它是返回子对象的引用。例如，下面程序段得到脸上左眼的对象引用，然后将其改成黑色：

```
Figure eye = face.child(1);
eye.setPainter(new FillPainter(Color.black));
```

同时，`GroupNode` 提供了按索引顺序遍历子节点的迭代器。用下面这段代码可以将 `face` 的所有子节点的轮廓线改为绿色：

```
Iterator iter = face.iterator();
while (iter.hasNext()) {
    Figure fig = (Figure)iter.next();
    fig.setPainter(new DrawPainter(Color.green));
}
```

到目前为止，我们所讨论的是图 6-22 的左边情景图，它共有两层深。前面讲过，`GroupNode` 是可以嵌套的，所以在图 6-22 右边的 `anotherFace` 情景图中，用三层表示：根 `anotherFace` 的 `eyes` 是 `GroupNode`，它由 `leftEye` 和 `rightEye` 组成，下面代码段生成了情景图 `anotherFace`：

```
// build the composite figure eyes
GroupNode eyes = new GroupNode();
eyes.addChild(leftEye);
eyes.addChild(rightEye);
// build the composite figure anotherFace
GroupNode anotherFace = new GroupNode();
anotherFace.addChild(contour);
anotherFace.addChild(eyes);
anotherFace.addChild(nose);
anotherFace.addChild(mouth);
```

`GroupNode` 的子节点可以有两种类型：`GroupNode` 或 `Figure`。这两种类型都具有画图功能，所以它们有相同的父型，称其为 `Node`。`Node` 接口定义如下：

```
public interface Node {
    public void paint(Graphics2D g2);
    // REQUIRES: g2 is not null.
    // EFFECTS: Paints this scene graph into
    // the rendering context g2.
}
```

因为前面 `Figure` 类的定义已经实现过 `paint` 方法，所以 `Figure` 类的定义只需要修改头，以便声明它实现 `Node` 接口：

```
public class Figure implements Node {
    // same as before
    ...
}
```

同样即将定义的 `GroupNode` 类也要实现 `Node` 接口。

图 6-23 是 `Node`、`GroupNode` 和 `Figure` 之间的类关系图。图中说明 `GroupNode` 由一

组Node对象组成，也称这些Node为GroupNode的孩子。GroupNode和Node之间是聚集关系，Node类边上的多重性标志“*”表示0或多个：GroupNode可以包含0个或多个Node对象。Figure类表示原子组件。Node接口提供了所有组件共享的接口，无论是组节点还是图形。

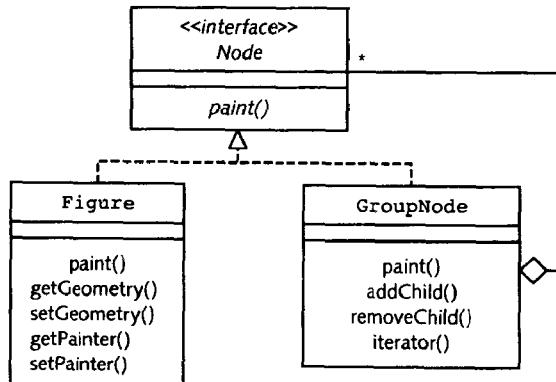


图6-23 组合图的类图

虽然GroupNode类和Figure类都要实现Node接口，但它们的实现是不同的。如图6-23所示，GroupNode除了提供了增加、删除、访问子组件的方法外，还提供一些其他的方法；图形提供设置和获得geometry和painter特性的方法。在组合设计模式中，尽管所有组件都共享接口Node，但不论是原子组件（如Figure）和组合体（GroupNode）都可以扩展自己的接口。

下面将介绍GroupNode的实现代码，其中GroupNode的子节点保存在矢量中按索引排序，下面是实现：

```

public class GroupNode implements Node {
    protected Vector children;

    public GroupNode() {
        // EFFECTS: Creates a group node with no children.
        children = new Vector();
    }

    public int nbrChildren() {
        // EFFECTS: Returns the number of children.
        return children.size();
    }

    public void addChild(Node node, int i)
        throws NullPointerException,
               IndexOutOfBoundsException {
        // MODIFIES: this
        // EFFECTS: If node is null throws
        // NullPointerexception; else if 0 <= i and
        // i <= nbrChildren() inserts node as the i'th
        // child and increases by one the index of every
        // child whose index not less than i; else
        // throws IndexOutOfBoundsException.
        if (node == null) throw new NullPointerException();
        children.add(i, node);
    }
}

```

```

}

public void addChild(Node node)
    throws NullPointerException {
    // MODIFIES: this
    // EFFECTS: If node is null throws
    // NullPointerException; else inserts node
    // as the last child.
    addChild(node, children.size());
}

public void removeChild(int i)
    throws IndexOutOfBoundsException {
    // MODIFIES: this
    // EFFECTS: If 0 <= i < nbrChildren() removes the
    // i'th child and decreases by one the index of
    // every child whose index is greater than i;
    // else throws IndexOutOfBoundsException.
    children.remove(i);
}

public Node child(int i)
    throws IndexOutOfBoundsException {
    // EFFECTS: If 0 <= i < size() returns the i'th child;
    // else throws IndexOutOfBoundsException.
    return (Node)children.get(i);
}

public Iterator iterator() {
    // EFFECTS: Returns an iterator which visits this
    // node's children in index order.
    return children.iterator();
}

public void paint(Graphics2D g2) {
    // REQUIRES: g2 is not null.
    // EFFECTS: Paints this scene graph into g2.
    Iterator iter = iterator();
    while (iter.hasNext()) {
        Node node = (Node)iter.next();
        node.paint(g2);
    }
}
}

```

GroupNode类中的大部分方法都是委托它的矢量children实现的，观测paint方法的实现可以发现它可以处理任何子节点，只要发送一个paint消息给node，不管它是组节点还是图形。

6.4.2 建立坐标轴

在本节中，我们将定义一个平面上的坐标轴类Axes，这主要是为在下一节介绍坐标系打基础。大家都知道，一对坐标轴是由两条线组成：水平轴x和垂直轴y，每条轴上有均匀的标记单位量。如图6-24所示，水平轴和垂直轴的相交点是(0, 0)，基本单位量是50(脸的半径是100)。右边的情景图说明，一对坐标轴是由一个组节点表示的，它包含两个子节点xAxis和yAxis，这两个也是组节点，它们各自由一组Figure对象组成，其中包括坐标线和标记。

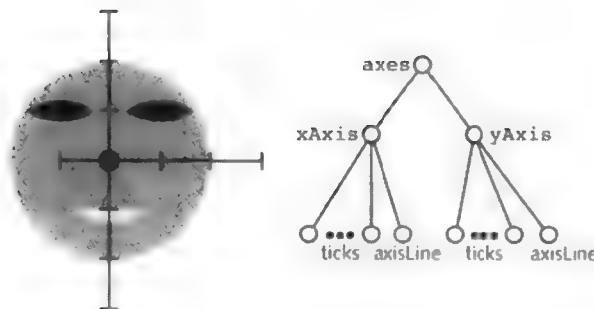


图6-24 一对坐标轴的情景图

带有四个参数的Axes类构造器如下：

```
public Axes(Range xRange, Range yRange,
           int tickStep, Painter painter)
```

参数xRange确定x轴的变化：假定xRange的变化范围是[xMin...xMax]，x轴从点(xMin, 0)到点(xMax, 0)；y轴与此相同；参数tickStep表示坐标单位量；参数painter表示一个Painter对象的引用，用于画坐标轴。下面代码段将产生图6-24中的坐标轴：

```
Range xRange = new Range(-50, 150);
Range yRange = new Range(-150, 150);
Painter black =
    new DrawPainter(Color.black, new BasicStroke(2));
Node axes = new Axes(xRange, yRange, 50, black);
```

要产生图6-24中的带坐标轴的脸，可以新建一个组节点，它包含两个子节点face和axes：

```
GroupNode faceAndAxes = new GroupNode();
faceAndAxes.addChild(face);
faceAndAxes.addChild(axes);
```

下面的代码Axes类中，tickStep特性表示坐标相邻两点之间单位数量，tickHeight特性表示坐标单位标记的高度。xAxis和yAxis分别表示x和y轴：

```
public class Axes extends GroupNode {

    protected static int DefaultTickHeight = 2;
    protected static Range DefaultRange =
        new Range(-100, 100);
    protected static int DefaultTickStep = 50;

    protected int tickStep, tickHeight = DefaultTickHeight;
    protected Range xRange, yRange;
    protected Painter painter;

    public Axes(Range xRange, Range yRange,
               int tickStep, Painter painter)
        throws NullPointerException, IllegalArgumentException {
        // EFFECTS: If xRange, yRange, or painter is null
        // throws NullPointerException; else if tickStep<=0
        // throws IllegalArgumentException; else constructs
        // a pair of axes whose x and y extents are given
        // by xRange and yRange, the interval between tick
        // marks is given by tickStep, and the axes are
```

```

// painted by painter.
if ((xRange==null)|| (yRange==null)|| (painter==null))
    throw new NullPointerException();
if (tickStep <= 0)
    throw new IllegalArgumentException();
this.tickStep = tickStep;
this.xRange = xRange;
this.yRange = yRange;
this.painter = painter;
createAxes();
}

public Axes(Painter painter)
    throws NullPointerException {
// EFFECTS: If painter is null throws
// NullPointerException; else constructs a pair
// of axes with x and y extents [-100..100], tick
// marks every 50 units, and painted by painter.
this(DefaultRange, DefaultRange,
    DefaultTickStep, painter);
}

public int getTickStep() {
    // EFFECTS: Returns the interval between tick marks.
    return tickStep;
}

public void setTickStep(int newTickStep)
    throws IllegalArgumentException {
// MODIFIES: this
// EFFECTS: If newTickStep <= 0 throws
// IllegalArgumentException; else sets the
// interval between tick marks to newTickStep.
if (newTickStep <= 0)
    throw new IllegalArgumentException();
tickStep = newTickStep;
createAxes();
}

public int getTickHeight() {
    // EFFECTS: Returns the height of a tick mark.
    ...
}

public void setTickHeight(int newTickHt)
    throws IllegalArgumentException {
// MODIFIES: this
// EFFECTS: If newTickHt is negative throws
// IllegalArgumentException; else sets the height
// of tick marks to newTickHt.
    ...
}

protected void createAxes() {
// REQUIRES: Instance fields are initialized.
// MODIFIES: this.children
// EFFECTS: Creates a new set of axes.
if (nbrChildren() >= 2) {
    removeChild(1);
    removeChild(0);
}
GroupNode xAxis = createXAxis();
GroupNode yAxis = createYAxis();
}

```

```

        addChild(xAxis, 0);
        addChild(yAxis, 1);
    }

protected GroupNode createXAxis() {
    // REQUIRES: Instance fields are initialized.
    // MODIFIES: this.children
    // EFFECTS: Creates a new x axis.
    GroupNode xAxis = new GroupNode();
    for (int x = xRange.getMin(); x <= xRange.getMax();
         x += tickStep) {
        Geometry tick =
            new LineSegmentGeometry(0, tickHeight,
                                   0, -tickHeight);
        tick.translate(x, 0);
        xAxis.addChild(new Figure(tick, painter));
    }
    Geometry axisLine =
        new LineSegmentGeometry(xRange.getMin(), 0,
                               xRange.getMax(), 0);
    xAxis.addChild(new Figure(axisLine, painter));
    return xAxis;
}

protected GroupNode createYAxis() {
    // REQUIRES: Instance fields are initialized.
    // MODIFIES: this.children
    // EFFECTS: Creates a new y axis.
    ...
}
}

```

注意，保护型createAxes方法是用于创建或重新创建坐标轴。如果坐标轴已经存在，tickStep或tickHeight发生变化时要重新创建，那么createAxes方法先删除已有的，然后再重新创建。

练习

6.28 (重点) 完成类Axes的实现，并编写程序产生图6-24中带坐标的脸。
 6.29 上面所定义坐标轴没有标注轴X、Y，请给类Axes加上增加和删除X、Y标注的方法：

```

public void addLabels()
    // MODIFIES: this
    // EFFECTS: If the axes are currently labeled does
    // nothing; else adds the labels X and Y to this set
    // of axes, rendered using TextGeometry.DefaultFont.

public void addLabels(Font font)
    // MODIFIES: this
    // EFFECTS: If the axes are currently labeled does
    // nothing; else if font is null throws
    // NullPointerException; else adds the labels X and
    // Y to this set of axes, rendered using font.

public void removeLabels()
    // MODIFIES: this
    // EFFECTS: If the axes are not currently labeled does
    // nothing; else removes the labels from the axes.

```

带有X、Y标注的坐标轴将在图6-26中出现。你也许会考虑用Figure对象表示X、Y标注，它的几何图形是TextGeometry对象。例如，X标注将包含这一几何图形：

```
new TextGeometry(font, "X")
```

其中font将用于addLabels方法中（如果没有传入参数font，则使用TextGeometry的默认字体），这样X、Y标注就可以如同子组件一样加入和删除。还有一个问题，最好定义一个boolean域，它的值用于标识坐标是否已经标注。

6.30 编写程序PaintFlower，输出如图6-21的花朵图形：

```
> java PaintFlower n radius
```

输出的花朵有n个叶子，它们均匀分布在中间圆的周围。圆的半径为radius，叶子要有比例，它们的长度应该超过圆的直径，但不能超出太多。

6.4.3 可变换组合图

当一个图形对象输出图形到Java绘图环境（一个Graphic2D对象）时，它们使用用户空间坐标系定义。默认情况下，用户空间坐标系和Java的默认坐标系是一致的。它们的坐标原点在屏幕的左上角，x轴向右扩展，y轴向下扩展，并且x和y都为像素值。用户空间坐标系是由Java绘图环境中的transform属性决定的，所以它是可以改变的。实际上，前面已经有这样的例子，在3.3.4节中修改过用户空间坐标系。

在本节中，我们将定义一个封装坐标系的组节点类型。TransformGroup类是GroupNode的子类，它定义了其子节点定位的坐标系。为什么要定义新坐标系呢？有三个原因：第一，可以按我们的意志定位图形，有时直接使用默认坐标系不行。例如，在默认坐标系下，如果一个矩形的边平行于坐标系，则表示矩形的边平行于画图平面的边。在旋转坐标系下，如果矩形的边平行于坐标轴，则表示矩形也是旋转的（见图6-26）。

第二，为了提高效率，我们常常定义多个坐标系。在多个坐标系中，一个节点的定位可以通过创建多个引用实现，每个引用都相对于它的坐标系，与其他引用无关。这样可以避免定义多个节点的麻烦，使情景图小而清晰。更重要的是，这使节点的修改更容易：当节点被修改时，如改变颜色或增加子节点，则它的每个引用自动跟着变化。例如，如果给图6-30的一个脸加上耳朵，则其他25张脸都有耳朵。

第三，可以选择一个易用的坐标系画图，而不用考虑所绘图形和其他图形的关系。比如说画草地上的一朵花，可以选择它的花心为坐标原点，草地上其他花朵也如此。完成后所有的花朵都可以依草地的坐标定位。

1. TransformGroup类行为

TransformGroup类是GroupNode的子类，它比父类多定义了一个新的坐标系，用于定位它的子节点。它们之间的关系见图6-25。任何TransformGroup或GroupNode的实例统称为分组节点（Grouping node）。

在情景图中，TransformGroup新定义的坐标系是相对于它的父节点坐标系的，新坐标系称为子坐标系（child coordinate system），它被嵌入到它的父坐标系（parent coordinate system）。子坐标系和父坐标系是相对的，一个父坐标系相对它的父坐标系又称为子坐标系。这样情景图可以看作是坐标系的层次图，根节点定义了自己的坐标系，下

面的子节点都相对它的父节点定义自己的坐标系，叶节点仍然是Figure对象。

由于TransformGroup的子节点是依据它的坐标系定位的，所以当坐标系发生变化时，如移动、缩放、旋转，它的子节点同样变化。TransformGroup类提供了几个坐标变化的操作（见图6-25），下面将举例来看它们的用法。这些例子中都假设变量squareFig引用Figure对象，表示一个边长为100、中心点在坐标原点的正方形，该图可以构造如下：

```
Geometry geom = new RectangleGeometry(-50, -50, 100, 100);
Painter fillPainter = new FillPainter(Color.green);
Painter drawPainter = new DrawPainter(Color.red);
Painter painter =
    new MultiPainter(fillPainter, drawPainter);
Figure squareFig = new Figure(geom, painter);
```

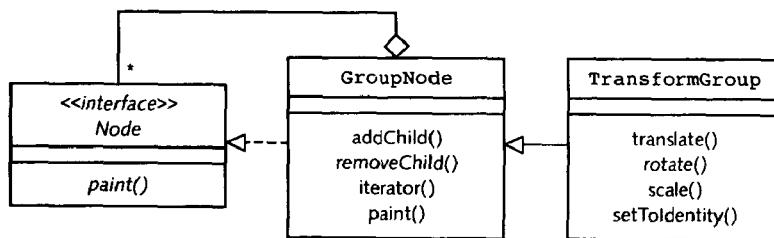


图6-25 TransformGroup节点是定义了局部坐标系的分组节点

下面代码创建TransformGroup对象rNode，加入squareFig为子节点，然后rNode的坐标系顺时针旋转45°：

```
TransformGroup rNode = new TransformGroup();
rNode.addChild(squareFig);
rNode.rotate(45.0);
```

图6-26中第一个图表示了画rNode到绘图环境g2的结果：

```
rNode.paint(g2);
```

图中有两对坐标轴：黑色坐标轴是父坐标系；灰色坐标轴是由rNode定义的，正方形所在的坐标系。

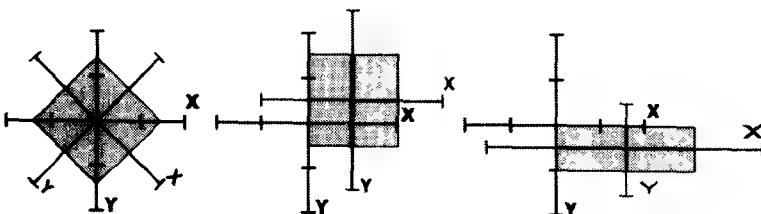


图6-26 旋转坐标系、移动坐标系和缩放移动坐标系中的正方形

同样可以移动坐标系，下面代码产生如图6-26中第二个图所示的结果：

```
TransformGroup tNode = new TransformGroup();
tNode.addChild(squareFig);
tNode.translate(50, -25);
tNode.paint(g2);
```

坐标系缩放是指坐标系的水平或垂直轴的单位成比例放大或缩小，图形相应地跟着变化。水平轴和垂直轴的变化是互不干涉的，它们各自按自己的比例缩放，下面的代码段

产生图6-26中的第三个图：

```
TransformGroup sNode = new TransformGroup();
sNode.scale(1.5, .5);
sNode.translate(50, 50);
sNode.addChild(squareFig);
sNode.paint(g2);
```

上面坐标系进行了两次变化：先缩放，再移动。需要注意的是坐标X、Y标识大小也随着变化；另一个与前面程序不同之处是sNode先变化坐标，后增加子节点squareFig。一般来说，两者的顺序是没有要求的。

显然，可以把刚刚定义的rNode等加入到普通的GroupNode中，下面把rNode、tNode作为GroupNode对象的子节点：

```
GroupNode scene = new GroupNode();
scene.addChild(rNode);
scene.addChild(tNode);
```

结果是一个根节点scene带有子节点rNode和tNode的情景图。当输出到绘图环境g2时：

```
scene.paint(g2);
```

该情景图显示同一正方形两次，每个正方形独立定位和定向。图6-27说明了情景图为什么不称为情景树（scene tree）的原因：因为一个子节点有可能属于两个或多个父节点。如本例中squareFig同时属于rNode和tNode。实际上，情景图的结构称作为直系非循环图（directed acyclic graph），是一种广义的树状图。注意，squareFig的任何变化都影响rNode、tNode的结果。例如，下面一条语句将使图6-27的两个正方形变为洋红色：

```
squareFig.setPainter(new FillPainter(Color.magenta));
```

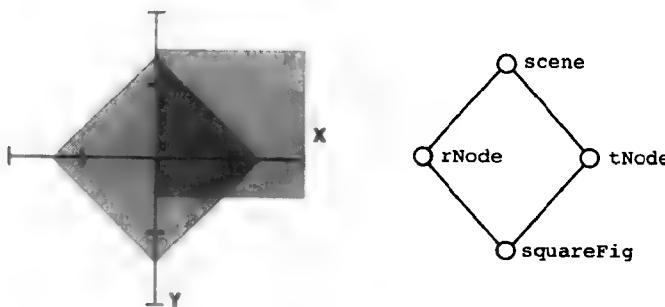


图6-27 情景图scence和它的输出图形

根据上面的描述，可以得出TransformGroup类的类框架：

```
public class TransformGroup extends GroupNode {

    public TransformGroup()
        // EFFECTS: Constructs a new transform group having
        // the same coordinate system as its parent node
        // in the scene graph.

    public void rotate(double theta)
        // MODIFIES: this
        // EFFECTS: Rotates this coordinate system by theta
        // degrees around its origin, from the x axis
        // toward the y axis (i.e., clockwise in the
```

```

// default user space).

public void rotate(double theta, PointGeometry center)
    throws NullPointerException
// MODIFIES: this
// EFFECTS: If center is null throws
//    NullPointerException; else rotates this
//    coordinate system by theta degrees around
//    center, from the x axis toward the y axis.

public void scale(double sx, double sy)
    throws IllegalArgumentException
// MODIFIES: this
// EFFECTS: If sx or sy equals zero throws
//    IllegalArgumentException; else scales this
//    coordinate system by sx along x and sy along y.

public void translate(double dx, double dy)
// MODIFIES: this
// EFFECTS: Translates this coordinate system by dx
//    along x and dy along y.

public void setToIdentity()
// MODIFIES: this
// EFFECTS: Restores this coordinate system to that
//    of its parent node in the scene graph.

public void paint(Graphics2D g)
// EFFECTS: Paints this node's children in index
//    order within its coordinate system.
}

```

2. 图形变换方法和绘图环境

在我们了解 `TransformGroup` 类的详细实现之前，需要一些预备知识。正如我们已经看到的，变换组根据另一个坐标系，特别是其父节点的坐标系，指定一个新的坐标系。组节点通过坐标系变换（coordinate-system transformation）简称为变换（transform）的方式表示它的坐标系，变换得到不同的两个坐标系，然后从原坐标系经过必要的步骤到达目标坐标系。

Java 中提供了坐标变换类 `java.awt.geom.AffineTransform`。`AffineTransform` 的无参数构造器创建并返回一个新坐标对象，同原来的坐标系完全相同。坐标系变换是由它的 `translate`, `rotate`, `scale` 等操作实现的。下面的代码段将使 `tform` 的坐标按顺时针旋转 45°，然后沿 `x` 轴平移 50 个单位，沿 `y` 轴平移 100 个单位：

```

AffineTransform tform = new AffineTransform();
tform.rotate(Math.toRadians(45));
tform.translate(50, 100);

```

`AffineTransform` 类还提供了 `setToIdentity` 方法，恢复到原来的坐标系。下面的 `AffineTransform` 类的类框架只说明了我们需要的那些方法：

```

public class AffineTransform {

    public AffineTransform()
        // EFFECTS: Initializes this to identity transform.

    public void translate(double dx, double dy)
}

```

```

// MODIFIES: this
// EFFECTS: Translates this by dx along x and
//           dy along y.

public void rotate(double theta)
    // MODIFIES: this
    // EFFECTS: Rotates this by theta radians around
    //           the origin, from the x axis toward the y axis.

public void rotate(double theta, double x, double y)
    // MODIFIES: this
    // EFFECTS: Rotates this by theta radians around
    //           (x,y), from the x axis toward the y axis.

public void scale(double sx, double sy)
    // MODIFIES: this
    // EFFECTS: Scales this by sx along the x axis and
    //           sy along the y axis.

public void setToIdentity()
    // MODIFIES: this
    // EFFECTS: Restores this to the identity transform.
}

```

我们前面已经使用过`AffineTransform`的坐标变换功能。例如在3.4.4节中, `Graphics2D`对象有一个`transform`属性确定绘图发生的坐标变换。`transform`属性的值是`AffineTransform`对象。前面的某些程序使用`transform`属性调整输出图形相对于框架的位置。考虑下面的过程`paintComponent`, 它画一个节点`scene`到绘图环境`g2`:

```

public void paintComponent(Graphics g) {
    super.paintComponent(g);
    Graphics2D g2 = (Graphics2D)g;
    Dimension d = getFrame().getContentSize();
    g2.translate((int)(d.width/2), (int)(d.height/2));
    scene.paint(g2);
}

```

`g2`的原始坐标原点是在左上角, 发送坐标移动消息给`g2`后, `g2`的坐标系的原点移动到框架的中心点。这样绘图时就从框架中心位置开始。`g2`的`transform`属性的值是由发送的`translate`消息修改的。语句

```
g2.translate(50,100)
```

的完成过程是: `g2`收到`translate`消息后, 发送`translate`消息给它的`AffineTransform`对象。

可以猜想, `Graphics2D`类同样会提供其他几种修改坐标系方法。`scale`方法用于缩放坐标系, 沿x轴缩放`sx`, 沿y轴缩放`sy`:

```
// method of Graphics2D class
public void scale(double sx, double sy)
```

`rotate`方法用于旋转坐标系, 沿 (x,y) 坐标点顺时针旋转`theta`度

```
// method of Graphics2D class
public void rotate(double theta, double x, double y);
```

例如, 下面的代码段说明如何在框架中央输出一个旋转 45° 的正方形:

```
Dimension d = getFrame().getContentSize();
```

```

g2.translate((int)(d.width/2), (int)(d.height/2));
g2.rotate(Math.toRadians(45), 0, 0);
squareFig.paint(g2);

```

这里g2的坐标系先被移动到框架的中心，然后绕框架中心的原点旋转45°。

Graphics2D类还提供了一个transform方法，它以AffineTransform对象为输入参数，并将其应用到它的坐标系：

```

// method of Graphics2D class
public void transform(AffineTransform tform)

```

所以另一种改变g2坐标系的方法是：定义合适的AffineTransform对象tform并调整它的坐标系，然后把tform传给g2，修改属性transform值为tform。例如下面代码的结果是：旋转g2的坐标系45°，然后沿x轴移50个单位，沿y轴移100个单位：

```

AffineTransform tform = new AffineTransform();
tform.rotate(Math.toRadians(45));
tform.translate(50, 100);
g2.transform(tform);
squareFig.paint(g2);

```

Graphics2D类还提供了两个方法完成设置和获得属性transform，它们是：

```

// methods of Graphics2D class
public void setTransform(AffineTransform newTform)
public AffineTransform getTransform();

```

总之，Graphics2D类的实例（绘图环境）有transform属性和一组坐标系变化方法。transform属性的值是java.awt.geom.AffineTransform对象，它指定了客户使用绘图环境绘图时所用的坐标系。绘图环境提供了大量用于修改其坐标系的方法，如移动、缩放、旋转、恢复，以给定的变换进行变换等。无论绘图环境何时接收任何类型的变换消息，它都以调用的方式变换它的AffineTransform对象。

3. 实现TransformGroup类

我们终于要实现TransformGroup类了！TransformGroup类是GroupNode的子类，它比父类GroupNode多定义了一个新的坐标系，用于定位它的子节点。如同Graphics2D一样，会考虑用到AffineTransform类的实例。实际上，AffineTransform类作为组件同时用在这两个类，它们三者的关系见图6-28：

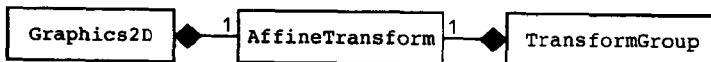


图6-28 TransformGroup对象和Graphics2D对象都用AffineTransform表示它的坐标系的关系

下面先看TransformGroup类的域定义：

```

// field of TransformGroup class
protected AffineTransform tform;

```

TransformGroup类只定义了一个无参数构造器，方法中创建了AffineTransform对象tform。这样在情景图中，TransformGroup对象和它父节点有相同的坐标系：

```

public TransformGroup() {
    tform = new AffineTransform();
}

```

在TransformGroup对象的生存期中，它可能收到不同的坐标系变化消息，如

translate、rotate或scale。这些消息是通过授权完成的：TransformGroup对象将这些消息转发给存储在tform域中的AffineTransform组件。下面是TransformGroup的方法定义：

```
// methods of TransformGroup class
public void rotate(double theta) {
    tform.rotate(Math.toRadians(theta));
}

public void rotate(double theta, PointGeometry center)
    throws NullPointerException {
    tform.rotate(Math.toRadians(theta), center.getX(),
                center.getY());
}

public void scale(double sx, double sy)
    throws IllegalArgumentException {
    if ((sx == 0) || (sy == 0))
        throw new IllegalArgumentException();
    tform.scale(sx, sy);
}

public void translate(double dx, double dy) {
    tform.translate(dx, dy);
}

public void setToIdentity() {
    tform.setToIdentity();
}
```

注意，旋转的输入参数是度，不是弧度。

TransformGroup类的paint方法看起来要复杂一些。下面是paint方法的实现：

```
// method of TransformGroup class
public void paint(Graphics2D g2) {
    AffineTransform oldTform = g2.getTransform(); // step 1
    g2.transform(tform); // step 2
    super.paint(g2); // step 3
    g2.setTransform(oldTform); // step 4
}
```

paint方法的执行有四步，其中g2是传给paint的绘图环境：

- 1) 获得并保存g2的变换的拷贝，这个保存的变换相当于情景图中父节点的坐标系。
- 2) 把本节点的变换tform应用到存储在g2中的变换，在这一步完成时g2表示这个变换组的坐标系。
- 3) 输出这一节点的子节点到g2。因为TransformGroup的父类是GroupNode类，方法调用super.paint(g2)；按索引顺序输出子节点。
- 4) 恢复g2的前一个变换，这是必须的，因为调用paint方法的客户可以假设g2的坐标系没有变化。

在情景图的图形输出处理过程中，绘图环境g2负责把父节点的坐标系转达给它的每个子节点，这也说明为什么第4步中恢复g2的坐标系对整个情景图非常重要。每个节点都负责按它的坐标系输出它的子节点，因此它要通过绘图环境g2把它的坐标系传给每个子节点，而它的子节点要确保g2的坐标系和传入时一样，亦即g2的transform属性没有变化。

下面看一个例子，看一下如何用TransformGroup节点建立定位在不同的坐标系上

的输入node的情景图。LineOfNodes类表示一个带有n个子节点的组节点，每个子节点是TransformGroup对象，TransformGroup对象又包含一个node子节点。这n个TransformGroup节点的坐标是按一定的规律移动的：第1个的坐标原点移动到点pos，以后每一个都相对于上一个移动dx和dy，下面是该类的实现：

```
public class LineOfNodes extends GroupNode {
    public LineOfNodes(Node node, int n, PointGeometry pos,
                       int dx, int dy)
        throws NullPointerException {
        for (int i = 0; i < n; i++) {
            TransformGroup t = new TransformGroup();
            t.addChild(node);
            t.translate(pos.getX() + i*dx, pos.getY() + i*dy);
            addChild(t);
        }
    }
}
```

下面代码段使用LineOfNodes类产生有五张脸的图形，其中face是Figure节点，半径为25，坐标系的单位间隔为50，见图6-29：

```
faces =
    new LineOfNodes(face,5,new PointGeometry(0,0),50,25);
faces.paint(g2);
```

节点faces的5个子节点都是TransformGroup，每个TransformGroup节点又有一个子节点face。图6-29右边是对应的情景图。

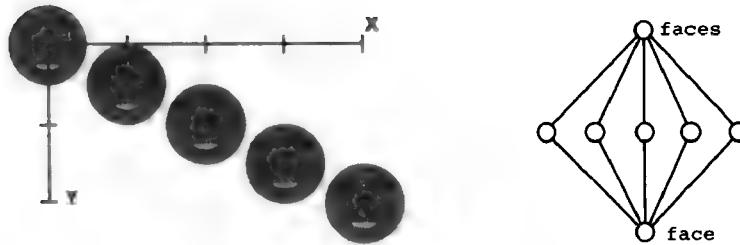


图6-29 组节点faces和5个TransformGroup子节点，每个子节点包含face作为它的子节点

练习

6.31 先考虑下面类的目的是什么？

```
public class MatrixOfNodes extends GroupNode {
    public MatrixOfNodes(Node node, PointGeometry pos,
                         int n1, int dx1, int dy1,
                         int n2, int dx2, int dy2)
        throws NullPointerException {
        Node line =
            new LineOfNodes(node, n1, pos, dx1, dy1);
        Node matrix =
            new LineOfNodes(line, n2, pos, dx2, dy2);
        addChild(matrix);
    }
}
```

- 1) 写一个使用类MatrixOfNodes的程序，该程序产生图6-30的图形，其中

`face`是代表半径为20的脸形的Figure对象。用以下语句可以产生图6-30所示的脸形图矩阵：

```
manyFaces =
    new MatrixOfNodes(face, new PointGeometry(0, 0),
                      5, 50, 0, 5, 25, 50);
```

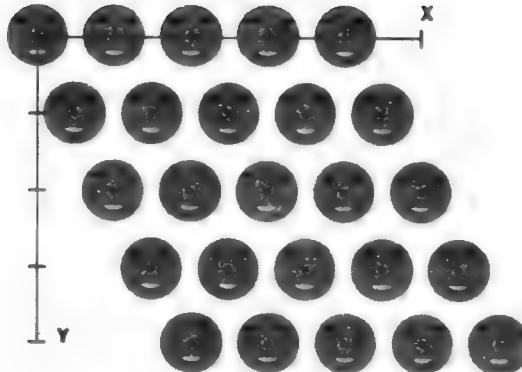


图6-30 脸形矩阵图

2) 画出以`manyFaces`节点为根的情景图。

6.32 下面类`ScaleAndRotateGroup`用于表示包含`n`个被旋转和缩放的输入`node`的实例，其中第一个`node`实例没有变换，其他的`node`实例是在它上一个实例的基础上经过缩放(`sfx, sfy`)和旋转`theta`度后得到。这说明它的结果，实际上并没有描述它的构造器如何工作和情景图是如何产生的：

```
public class ScaleAndRotateGroup
    extends TransformGroup {
    public ScaleAndRotateGroup(Node node, int n,
                               double sfx, double sfy, double theta)
        throws NullPointerException {
        if (n == 1)
            addChild(node);
        else {
            TransformGroup tgroup =
                new ScaleAndRotateGroup(node, n-1,
                                       sfx, sfy, theta);
            tgroup.scale(sfx, sfy);
            tgroup.rotate(theta);
            addChild(tgroup);
            addChild(node);
        }
    }
}
```

1) 请画出下面的调用语句的情景图，以帮助理解上面的内容。

```
new ScaleAndRotateGroup(node, 3, .5, .5, 45.0);
```

2) 编写一段Java图形程序，绘出一组被缩放和旋转的规则多边形，程序调用如下：

```
> java PaintScaleAndRotate nbrSides radius n sfx sfy theta
```

参数`nbrSides`和`radius`描述作为节点传给`ScaleAndRotateGroup`构造器的初始规则多边形，其他四个参数是传给`ScaleAndRotateGroup`的构造器的四个附加参

数，下面这段代码可能构造出由通用代码行描述的情景：

```
Geometry polyGeom =
    new RegularPolygonGeometry(nbrSides, radius);
Painter painter =
    new DrawPainter(Color.black, new BasicStroke(4));
Figure polyFig = new Figure(polyGeom, painter);
Node scene =
    new ScaleAndRotateGroup(polyFig, n, sfx, sfy,
                           theta);
```

通过下面的三个调用，将依次产生图6-31中的三个图形：

```
> java PaintScaleAndRotate 4 100 8 .707 .707 45
> java PaintScaleAndRotate 4 100 16 .9 .9 10
> java PaintScaleAndRotate 8 100 40 .95 .8 10
```

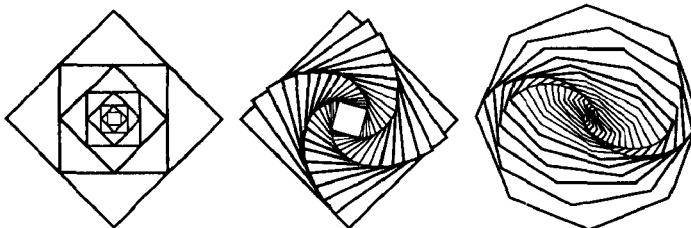


图6-31 程序PaintScaleAndRotate输出的图形

6.33 设计一个名为playSceneGraph的程序。它能进行命令解释并完成相应的图形创建和变换。用户使用define命令创建一个命名图形，并在后面可以用名字引用该图形。用户用以下形式的命令创建和命名带有指定维数的矩形：

```
define name rectangle x y width height
```

椭圆以类似的命令形式创建和命名：

```
define name ellipse x y width height
```

当新图形被构造时，它被赋予当前颜色，无论何时画一个图形，图形都是以它被赋予的颜色填充。程序保持通过其名字选择的当前图形：

```
select name
```

变换命令仅适用于当前图形。例如，下面的命令形式：

```
rotate 45 10 20
```

将当前图形绕点(10, 20)顺时针旋转45°。

本程序可以显示到目前已经实现的所有的图形，也可以显示或隐藏它们的坐标系的坐标轴，下面是所有命令的功能解释：

- **define *id* [rectangle | ellipse] *x y width height*** 创建一个名为*id*图形，位于点(*x, y*)，带有指定的*width*和*height*。如果命名为*id*的图形已经存在，该图形将被新图形替换。图形被赋予当前颜色并成为当前图形。图形的颜色使用当前有效设置。
- **color *red green blue*** 更新当前颜色，新颜色由三个整数参数确定，参数范围为0≤*red, green, blue*≤255。
- **select *id*** 选择名为*id*的图形为当前图形，如果没有命名为*id*的图形，不进行任何操作。

- `rotate theta x y` 绕点 (x, y) 旋转当前图形 $theta$ 度。
- `translate dx dy` 移动图形， x 轴移动 dx ， y 轴移动 dy 。
- `axes [on|off]` 显示或关闭当前图形的坐标系的坐标轴。
- `quit` 退出程序。

图6-32为下面一段交互示例的输出结果：

```
> java PlaySceneGraph
? color 200 100 100
? axes on
? define a rectangle -50 -50 100 100
? rotate -45 0 0      // left image of Figure 6.32
? color 0 0 255
? define b ellipse -25 -50 50 100
? translate 50 0      // middle image of Figure 6.32
? select a            // right image of Figure 6.32
```

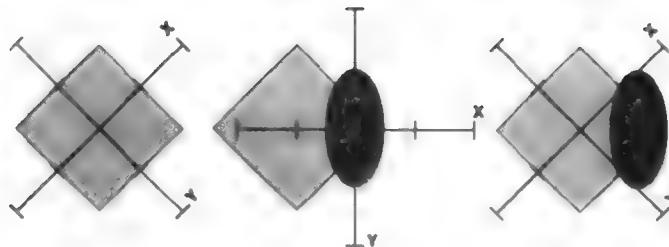


图6-32 程序playSceneGraph生成的图形输出

[提示：根节点 `scene` 类型为 `GroupNode`，`TransformGroup` 为它的子节点，`Figure` 为 `TransformGroup` 的子节点。显示图形就要发 `paint` 消息给 `scene`。另外，程序中要有一个到 `TransformGroup` 的引用以记录当前图形，因为旋转和移动图形是针对当前图形的。用程序的 `paintComponent` 方法显示坐标轴；如果坐标轴为可显示状态，使用这个方法增加一对坐标轴作为当前 `TransformGroup` 的子节点，然后画出 `scene`，然后删除该坐标轴。]

6.4.4 组合模式的结构和应用

组合设计模式用来构建原子组件和组合体的层次结构，使客户对原子组件和组合体能统一对待。这种设计模式主要是三个组成部分（如图6-33所示）：

- `Component` 接口 表示组合中的所有对象都支持的接口（如 `Node` 接口是 `Component`）。
- `Leaf` 类 每个 `Leaf` 类实现组件接口，它们的对象是最基本的原子组件。如 `Figure` 类是 `Leaf` 类。
- `Composite` 类 实现组件接口并定义对子组件的操作方法。如 `GroupNode` 是 `Composite` 类。

客户通过 `Component` 接口与对象进行交互。如果对象是 `Leaf` 类的实例，则它直接处理消息；如果对象是 `Composite` 类对象实例，则它要把消息转发给它的每个子组件，或许它自己还要做一些其他工作。如图6-33所示，所有组件都要实现组件接口中的 `op1`。组

合类中的方法 `addChild`、`removeChild` 和 `getChild` 表示管理子组件的操作。虽然本图中 `Leaf` 对象只有一个，但实际上可以有任意多个，并且增加 `Leaf` 并不影响其他已有的代码。特殊的组合体可以由已有的 `Composite` 类扩展得到，如同前面例子所示，`TransformGroup` 类扩展了 `GroupNode` 类。

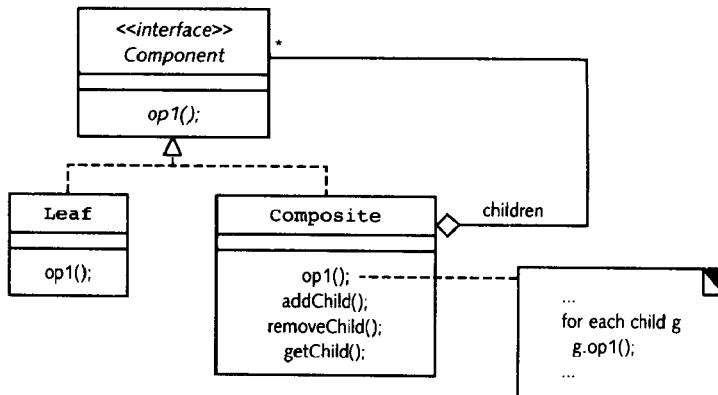


图 6-33 组合设计模式的结构

在组合设计模式中，所有的对象都实现相同 `Component` 接口大大地减少了客户的工作，因为它无需区分对象是叶子组件还是组合体。而当客户操作管理 `Composite` 对象的子组件或使用 `Leaf` 类或 `Composite` 类提供的其他方法时，就必须区分是叶子组件还是组合体。一些组合设计模式的变体把管理 `Composite` 的子组件的方法（如 `addChild` 和 `removeChild`）作为 `Component` 接口的一部分，这样做就把所有对象的接口都加大。这是以牺牲安全为代价的，因为这使得客户向 `Leaf` 对象发送毫无意义的子组件管理消息。

6.5 设计模式分类

设计模式种类繁多，并且新的设计模式不断涌现，所以需要恰当的分类方法对它们进行归纳。《设计模式》这本书使用两种分类方式。第一是根据目的对模式进行分类，即设计模式的目的是什么？设计模式的目的可划分为三种：创造型（*creational*）、结构型（*structural*）和行为型（*behavioral*）。创造型模式用于创建新的对象。结构型模式是为了解决如何用已有的类和对象组成更大的结构。行为型模式则注重于对象的职责的分配和对象间的协作关系。

第二种方式是按范围（*scope*）对模式进行分类，即设计模式是适用于类还是对象？类模式（*class pattern*）主要处理类、接口及其子型之间的关系。类模式是通过继承确立的，它是静态的，在整个程序的执行过程中是不变的。对象模式（*object pattern*）处理对象间的关系，对象模式是动态的，因为对象间的关系是随着程序的不断运行而变化的。

很多设计模式都应用这种思想：把系统的某个易变的部分独立出来，在改变它的时候并不影响系统的其他部分。这一点对理解设计模式是非常重要的。比如迭代器模式，我们可以定义不同的迭代器类用来按不同的顺序访问聚集中的元素（如逆序等），但客户使用不同的迭代器时并不受影响，因为每个都实现了客户要求的接口。在下面将要讨论的设计模式中，将会指出每种设计模式是如何封装它们的变化的。

回顾前面学习的三种设计模式，迭代器模式提供了一种顺序访问一个聚集对象中的各个元素的方法，而又不暴露它的实现或内部结构。因为它是关注对象之间的交互，即描述了迭代器对象如何同它的聚集对象和客户之间进行交互操作，所以它属于行为型模式。同时它又属于对象模式，因为迭代器和聚集之间不是基于继承而是基于对象之间的消息传递。迭代器模式是动态的：迭代器对象掌握着随时间变化的遍历状态，且新的迭代器可以随时建立。综上所述迭代器模式是一种行为对象模式。

模板方法模式定义一个算法，其中的一些算法步骤是由它的子类实现的。这就是说使用模板方法可以实现一个算法，这个算法的其中一些步骤随着不同的子类的不同实现而不同。模板方法模式属于行为类模式。它是类模式，这是因为它基于类之间的继承关系：抽象类定义模板方法并且它的子类实现模板方法所要求的抽象钩子方法。这些关系是静态的，因为实现同一算法不同部分的子类是在程序执行之前定义的。

组合模式用于将对象组合成原子组件和组合对象的层次结构，组合对象是由相对简单的原子对象组成的，但它们提供给客户的外部接口是一致的。这是一种结构对象模式，因为它提供了由简单对象形成复杂对象的方法。组合模式是动态的：在程序运行中允许有新的原子组件和新的组合体产生。

本节的其他部分将介绍和分析另外的几种常用的设计模式：工厂方法模式（factory method pattern）、适配器模式（adapter pattern）、观察者模式（observer pattern）、策略模式（strategy pattern）。

6.5.1 工厂方法模式

工厂方法模式用在客户调用一个方法创建一个新的对象而不知道要创建何种对象的时候，创建的具体对象由子类决定。工厂方法模式不仅能确保为客户提供一个统一的接口，而且使创建的对象的实际类型也可以变化。

举例来说，一个基于GUI的程序要在画布上画各种各样的图形。程序中的主要抽象工具是画图工具（paint tool），例如包括手工用的画笔、填充用的颜料桶、还有画曲线的工具。每种工具都由不同的类来实现，分别用BrushTool、BucketTool和CurveTool类表示。这些类都提供一个PaintTool接口以供程序使用。当用户点击程序工具条上一个按钮选择其中一个画图工具时，程序要创建一个对象表示选择的工具，可惜的是程序并不清楚所创建对象的实际类型。实际上，如果这个程序是一个可以随时增加新的画图工具的框架，那么它连到底支持哪些工具都不清楚。

工厂方法模式提供一种解决方法。不同按钮有不同的画图工具信息，每个按钮由抽象类ToolButton的子型的对象表示。抽象类ToolButton声明一个抽象方法createTool，它由每个子类进行实现。这样当程序得到用户点击某个画图工具按钮的事件时，它会保存当前工具按钮对象引用，并发送createTool消息给按钮对象，创建相应的PaintTool对象。举例来说，画笔按钮由一个BrushButton类的实例表示，它的createTool方法会创建一个BrushTool对象（见图6-34）。当用户点击画笔按钮时，程序收到这个事件后，把工具按钮对象引用保存到变量clickedButton中，并通过执行下面语句创建BrushTool类的实例：

```
PaintTool currentTool = clickedButton.createTool();
```

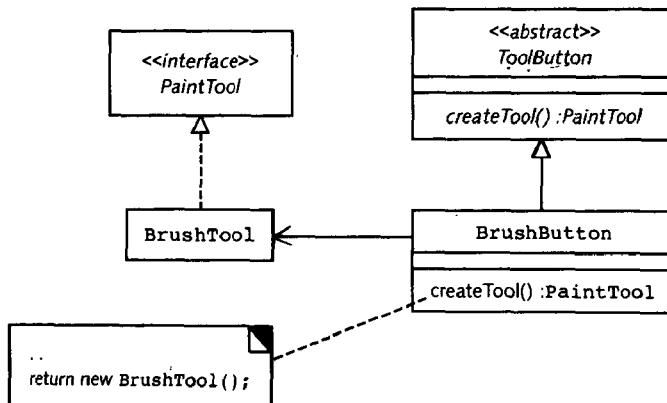


图6-34 工厂方法模式举例

由于对象创建被封装在`createTool`方法中，程序并不知道创建工具的实际类型。但因为`currentTool`对象实现`PaintTool`接口，所以程序可以在不知道它的实际类型的情况下使用它。

这里的`createTool`方法就称为工厂方法（factory method），因为它负责创建新的对象。在这种设计模式中，共有下面四种类型：

- **抽象Creator类** 指定工厂方法的类（如上面例中的`ToolButton`类就是一个`Creator`类）。
- **ConcreteCreator类** 扩展`Creator`类并实现它的工厂方法（如`BrushButton`和其他`ToolButton`类的子型）。
- **Product接口** 指定工厂方法创建的对象的接口（`PaintTool`）。
- **ConcreteProduct类** 每一个都实现`Product`接口；由工厂方法创建的对象是`ConcreteProduct`的实例（如`BrushTool`和其他`PaintTool`的子型）。

工厂方法模式降低了客户与实现它所使用的对象的类之间的依赖性。这就使得客户使用的类的修改更容易：客户通过工厂方法创建对象，对对象的处理通过它的表现类型`Product`而不是它的实际类型`ConcreteProduct`。同时对客户隐藏了要实例化哪一个类。

通常工厂方法由模板方法调用。为了实现一个算法，模板方法或许要求一个自己创建自己的新对象的服务，但对象的实际类型是由实现模板方法的子类决定的（每个子类依自己的方法实现算法）。这样模板方法就调用工厂方法来创建新对象，并且由子类决定创建何种对象。

工厂方法模式属于创造型模式，因为它用于创建新对象。同时它又属于类模式，因为它描述了类之间的继承层次关系。特别需要强调的是，父类（`Creator`）中定义的工厂方法可由一个或多个子类（`ConcreteCreator`）实现。`Creator`类的这种层次关系通常是和`Product`类的继承层次关系相平行对应的，这两个平行继承之间是通过工厂方法`createTool`的不同实现来沟通连接的。

6.5.2 适配器模式

适配器模式是为满足客户需求，把一个对象的接口转换成另一个客户需要的接口，使

得客户可以利用那些接口不兼容的对象的服务（这些对象称为服务对象）而不需要改变它们。也就是说，在客户和服务对象之间加上一个适配器，这个适配器将后者的接口适配为客户的需求，客户通过它来使用服务对象提供的服务。适配器的作用如同一个转换插头，在电源插孔和电器的插头的形状不相符时，使用一个转换插头将电源插孔和电器连接起来。

在这种设计模式中，**Adaptee**类提供了所需的功能但却是不恰当的接口，**Adapter**类提供客户需要的接口并访问**Adaptee**类的功能。下面是四个组成部分的描述：

- **Target**接口 表示客户希望的接口。
- **Client** 使用**Target**接口与对象进行交互。
- **Adaptee**类 提供客户所需的功能但是不提供需要与**Target**适配的接口。
- **Adapter**类 适配**Adaptee**类接口到**Target**接口。

从分类上说，适配器模式是属于结构型的，既可属于对象模式又可属于类模式。属于对象模式是因为**Adapter**对象包含**Adaptee**组件，**Adapter**对象把收到的消息转换成**Adaptee**组件可以理解的消息并传给它，由**Adaptee**对象负责完成实际的工作。在练习5.20中的**TextGeometry**类中就使用了这种设计模式。Java的**GlyphVector**类提供绘制图形文本的功能，但它却没有提供**Geometry**接口，所以设计了**TextGeometry**类，既为客户提供了**Geometry**接口又使用**GlyphVector**类的服务来实现它的形状操作。在图6-35中，**TextGeometry**扮演**Adapter**类角色，**GlyphVector**为**Adaptee**类，而**Geometry**接口是**Target**接口。

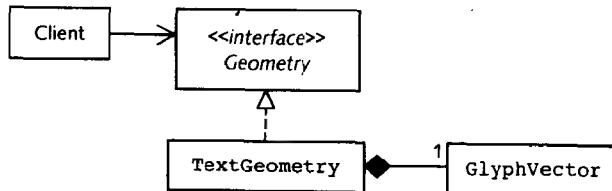


图6-35 适配器对象模式

从另一个方面说，适配器模式又属于类模式。这是因为**Adapter**类要继承**Adaptee**类的功能，是它的子类；同时**Adapter**类又要实现客户要求的**Target**接口。图6-36是从类的角度描述它们之间的关系，它们所扮演的角色同图6-35。

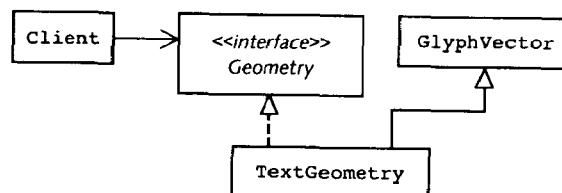


图6-36 适配器类模式

图6-36中所示的结构并不实用，这是因为Java的**GlyphVector**类是抽象类，需要**TextGeometry**类实现其中的抽象方法。在**Adaptee**类为具体类而非抽象类时，适配器类模式才更方便易用。适配器类模式的主要优点是**Adapter**类可以覆盖**Adaptee**类的某些行为；这种模式的主要缺点是**Adapter**类只能继承**Adaptee**类，却不能继承**Adaptee**

类的子类。与此相反，适配器对象模式却受益于**Adaptee**类的多态性：**Adapter**类的组件可以是**Adaptee**类的任何子类。

6.5.3 观察者模式

观察者模式适用于一个对象（称为目标）状态的变化对其他对象（称为观察者）的影响的情况下。当目标的状态发生变化时，它会通知每个观察者以便它们做出相应的响应。例如，目标为变化的数据，观察者为表示它们的方式，如电子数据表、柱状图、饼图等，当数据目标发生变化时，每个观察者都会在得到通知后及时更新表示。

目标要负责对观察者列表进行维护，提供增加新成员和删除已有成员的方法。当目标的状态发生变化时，它通过调用每个观察者的**update**方法通知它们。目标为封装其状态变化的**update**提供参数，或目标定义观察者用来查询其新状态的方法。这样或者目标将自身作为参数传给**update**或者观察者保存对目标的引用。

下面是这种设计模式涉及的四个组成部分：

- **Subject**类 负责对**Observer**列表进行维护，提供增加新观察者和删除已有的观察者的方法，而且通知**ConcreteSubject**中状态发生变化的观察者。
- **ConcreteSubject**类 扩展了**Subject**类，当它的状态发生变化时，通知所有观察者。
- **Observer**接口 定义了一个接口，**ConcreteSubject**的状态变化通过这个接口来通知**ConcreteObservers**。
- **ConcreteObserver**类 每个**ConcreteObserver**类都要实现**Observer**接口以提供响应**ConcreteSubject**状态变化的行为。

如图6-37所示，**ConcreteSubject**的状态发生变化时，它调用**notifyAll**方法，在此方法中依次通知它的每个观察者。

Java的事件模型是基于观察者模式。Java中组件如按钮、列表、面板等，用产生事件来响应用户的活动，如用鼠标点击或选择某一项，用户点击或选择它们时产生事件，事件监听者是定义了响应事件方法（称为事件处理器）的对象。在Java事件模型中组件是目标，事件监听者是观察者。当组件产生事件时，它会通知它的观察者，由它们来响应处理事件。Java的事件模型将在7.2节中详细介绍。

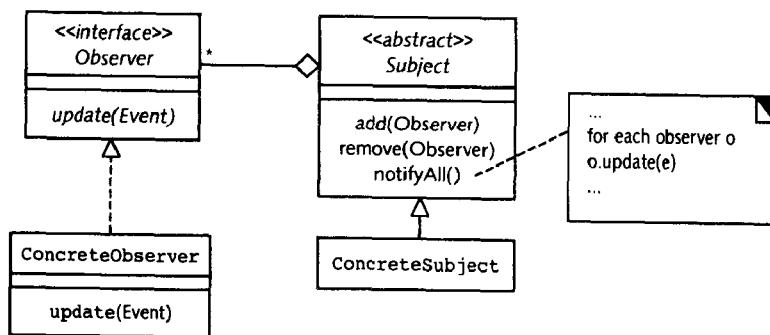


图6-37 观察者模式

观察者模式的最大优点是降低目标和观察者之间的相互依赖性。目标只需知道它有一组实现了**Observer**接口的观察者，而对它的观察者的具体实现一无所知。目标在它的状

态发生变化时负责通知观察者，由观察者完成响应变化。由此可知，这种模式允许系统两个方面的变化：观察者对象列表和它们响应目标变化的行为。

最后需要注意的是如同一个目标可以有多个观察者，一个观察者也可能同时观察多个目标。在这种情况下，当调用观察者update方法时，要（至少）依据传入参数确定哪个目标发生了变化。

6.5.4 策略模式

一个问题常常会有多种解决方法。策略模式正是把多种解决方法组织成一组对象，每个对象用一种不同的策略解决问题，但由于它们提供给客户统一的接口，客户可以选用它们中的任何一个。

举例来说，Java中的容器（如窗口）可以包含各种组件（如按钮、文本和列表等）。这些组件如何在容器中进行布局排列就应用不同的策略，称为布局管理器（LayoutManager），如FlowLayout把组件从左到右一行一行排列，而GridLayout按方格放置组件。当容器要排列它的组件时，它是通过LayoutManager的接口同它的布局管理器进行交互，所有布局管理器都要实现LayoutManager接口。容器通过一个通用接口来使用布局管理器，而无需了解布局管理器的实际策略。关于Java的布局管理器在7.4节中有介绍。

图6-38为Java布局管理器的策略模式，其中有三个部分：

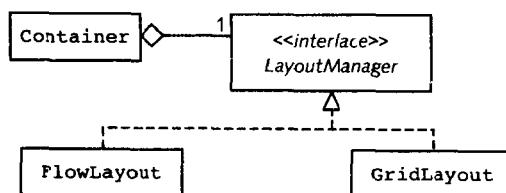


图6-38 用于组件布局的策略模式

- **Strategy**接口 定义所有ConcreteStrategy类支持的接口（如LayoutManager接口）。
- **ConcreteStrategy**类 每个这样的类都实现Strategy接口并实现它自己的其他策略（如FlowLayout和GridLayout）。
- **Context**类 包含一个ConcreteStrategy对象的引用，并通过这个引用调用Strategy接口执行一个策略（如Container）。

通常ConcreteStrategy需要Context的信息来完成它的任务，可有两种方法解决。一是Context调用策略时把所需的信息按参数传入；二是把整个Context按参数传入，这样策略可以通过直接查询Context得到所要的任何信息。Java的布局管理器是用第二种方法，当容器调用它的布局管理器时，把它自己作为传入参数，然后布局管理器就可以获得关于容器的任何信息，如容器的大小和它的组件的列表。

策略模式属于行为对象模式。它有下面几个优点：第一，客户如容器等的工作被简化，它只需调用通用的Strategy接口就可用不同的方法解决问题，解决问题的复杂算法由策略实现而不用客户处理。第二，运行时策略可以互换使用。第三，可随时增加新的策略，对客户毫无影响。

策略模式可以使一个算法独立于使用它的客户而变化。可以观察到策略模式与模板方法模式是用不同的方法达到相似的目标。策略模式是把解决问题的不同方法组合起来，利用委托来变换算法，而模板方法模式则是把解决问题的方法的某些变化步骤分离出来，通过子类继承来实现这些变化部分。

小结

简而言之，设计模式是一种反复出现的设计问题的解决方法。设计模式描述了所解决的问题和所使用的设计解决方法：使用的类和对象和它们之间的联系与协作。设计模式同时说明了使用这种模式的结果和利弊。设计模式是软件工程界经验和智慧的结晶，并且他们会继续致力于设计模式方面的工作，优化已有的设计模式并不断创造新的设计模式。

设计模式可按目的和范围分类。依据目的可分为三类：创造型模式用于创建新的对象；结构型模式解决如何用已有的类和对象组成更大的结构；行为型模式则注重于对象之间的协作关系。依据范围可分为两类：类模式主要处理类和接口以及子型之间的关系。对象模式强调对象间的关系。另外大多数设计模式都遵循系统的一个部分的变化并不影响其他部分的设计原则。

本章中介绍了下面几种常用的设计模式：

- 工厂方法模式是用在当客户要创建一个新的对象而又不知道要创建对象的实际类型的时候（创造型类模式）。
- 适配器模式是为满足客户需求，把一个对象的接口转换成一个客户可使用的接口（结构型对象模式或结构型类模式）。
- 组合模式是将对象组合成原子组件或组合体的层次结构，使客户能对原子组件和组合体一致对待（结构型对象模式）。
- 模板方法模式定义了一个算法，其中的一些算法步骤是由它的子类实现的（行为型类模式）。
- 迭代器模式提供了对一个聚集对象中的各个元素的访问，而又不暴露它的实现和内部结构（行为型对象模式）。
- 观察者模式定义了一组对象观察目标对象的状态的变化（行为型对象模式）。
- 策略模式是把多种策略的每一种封装起来，使它们通过公共接口对客户可用（行为型对象模式）。

第7章 面向对象应用程序框架

面向对象应用程序框架 (object-oriented application framework), 简称框架, 是设计重用的一种形式。框架是组成应用程序的类及接口的集合, 程序员可以按自己的意愿定制框架, 其目的是简化某个特殊领域的应用程序的开发过程。因为框架是和特定系统或编程语言联系在一起的, 因此它没有设计模式那么抽象。不过, 通常框架规模比较大, 而且结构单元中常常包含设计模式。

框架已经用于很多应用程序领域, 如多媒体、远程通信、操作系统、分布计算、商务系统及财务等。这一章, 我们将讨论抽象窗口工具 (Abstract Window Toolkit, AWT) 和 Swing, 它们组成Java的框架, 从而使我们利用图形用户界面 (graphical user interface, GUI) 来编程。7.1节是概述, 后面的部分以许多程序的开发为例深入介绍了Java的GUI框架。这一章的重点是用现代交互技术编写的可以绘制和编辑不同图形的程序。本章的目的是介绍创建GUI的Java框架, 并且利用这个框架来说明一般意义上的框架的特点。

7.1 用Java框架建立基于GUI的应用程序

本节综述框架的一般特点, 并且提供Java的框架的概述以便构造基于图形用户接口的程序。

7.1.1 框架的特点

面向对象的主要优点是对代码重用的支持。我们已经看到单个的软件组件可以通过继承和对象组合被重用, 而软件设计可以通过设计模式被重用。框架可以在更大的规模上支持重用。一个框架就是一个在特定应用程序领域的可重用的软件系统。如果采用框架来开发应用程序, 设计者只要定义与框架相联系的类便可以达到定制框架的目的。

对基于框架的应用程序而言, 框架本身像一个模具: 决定应用程序的总体设计, 并提供大部分或全部软件组件。甚至于当你为应用程序设计定制组件的时候, 框架也提供一般的组件, 便于在其基础上建立你的定制组件。这样你就可以将精力放在应用程序的行为上面, 其他就让框架来处理。

框架既支持设计重用, 也支持代码重用。支持设计重用是因为框架描述了你的应用程序的总体设计。通常说来, 这种设计暗含着控制倒置。当你从头开始开发应用程序的时候, 即便是利用类库或工具集, 你的应用程序控制执行流——它调用提供相应功能的对象。相反, 当你的应用程序是从一个框架继承而来的时候, 程序的执行流是由框架负责控制的。框架激活应用程序特定的行为, 而这些行为是由你, 即应用程序的设计开发者, 提供的。结果就是, 当你采用框架设计应用程序时, 你放弃了相当多的控制权。然而, 因为框架管理了一般来说会很复杂的执行流设计, 你的开发过程就变的简单省时, 而且最后的产品很可能更可靠, 更易于理解和维护。

框架重用代码是因为它们提供包含有用组件的库。框架提供的组件常常可以直接使用,

不必修改。同时，框架也简化了定制组件的开发。当需要特殊组件的时候，框架提供了可以用继承或组合方式来定制的通用组件。

框架定义了热点 (hot spot)，通过它开发者的代码和框架相连接的。每个热点规定了定制要遵循的规则。继承于相同框架的应用程序由于热点的存在而变得不同。应用程序采用继承或组合实现热点从而达到定制框架的目的。

当用继承的方法定制框架时，可以开发一个针对特定应用程序的类作为框架类。通过扩展Java的组件类来创建新的组件时可以用继承方法，比如说扩展JFrame或 JPanel类。新类继承了其父类的大部分功能，同时增加了新的方法和覆盖继承的一些方法以便形成独特的行为。用于定制的框架类常会提供钩子方法，便于它的子类覆盖。比如说，你可以定义一个扩展了 JPanel类并覆盖paintComponent方法的新类来完成任何绘图要求。要通过继承定义定制类，开发者应该对框架的继承层次结构非常熟悉。因为要求开发者必须理解被扩展类，所以有时称通过继承的定制为白盒 (white box) 机制。

框架也通过组合来定制。在这种情况下，框架为组件定义可以插入框架的接口。开发者定义新的组件或利用框架提供的组件来实现框架所要求的接口。通过组合的定制把程序开发者从框架的结构中解放出来，从而可以把精力集中在每个新组件的功能上。因为可以创建定制组件并重用存在的组件，而不必理解框架，所以通过组合的定制被称为黑盒 (black box) 机制。

很少有框架是严格的白盒或黑盒机制；大多数的框架同时用继承和对象组合来定制。这一点将会在用Java的框架来建立基于GUI的程序中得到证实。

框架和设计模式之间的关系值得关注。框架从规模上来讲通常比较大，并包含设计模式作为其结构单元，而且框架在特定的应用程序中还会引进某些特定的设计模式。在框架的热点中设计模式显得尤为重要，这是因为绝大多数的设计模式提供了保持系统其他方面不变的情况下改变系统某些方面的方法。通过继承的定制 (白盒) 常常采用模板方法设计模式来达到目的。框架为某算法定义一个模板方法，定制子类实现模板方法调用的抽象钩子方法，从而完成算法。通过组合的定制 (黑盒) 常用策略模式来达到目的。这里，框架定义一个由组件实现的接口以便组件可以被插入。开发者定义一个特制的组件来实现提供定制策略的接口。

由于框架是和系统联系在一起，并用某种编程语言来实现，所以它们没有设计模式那么抽象。从某个角度讲，框架比设计模式要大，因为它们包含设计模式。相反，设计模式独立于任何特定的编程语言。设计模式也是要被实现的，不过它们可在很多的情况下用任何一种语言实现。框架是一个程序，而设计模式是一种可由任何语言在任何程序中实现的抽象。

使用框架有很多优点，包括：

- 框架涵盖应用程序领域和应用程序编程的专门知识。通过重用，开发者可以利用这些专门知识而不用精通两者。
- 一般来说，用框架建立一个复杂的应用程序比从头开始写效率要高。
- 从框架继承而来的应用程序一般来说比较可靠，得益于开发框架时所投入的测试及精力。
- 基于框架的应用程序相对来说结构整齐。这意味着它们可能彼此相兼容，容易维护及扩展，并且倾向于提供相似的用户界面和交互约定。

使用框架也有许多不利之处：

- 建立一个好的框架很难，而且很费时间。
- 使框架随着时间而升级是一个必须进行而且艰难的过程。然而框架的升级通常又是不可避免的，因为应用程序的要求会改变、技术会发展、应用程序领域和框架的设计会有新的想法。
- 基于框架的应用程序会随着框架的升级而变化，以便利用新版本的特性和保持兼容。
- 定制一个框架使其完全符合应用程序的要求也许是不可能的。这样有时候就要修改应用程序的要求来迁就框架的能力，或加强框架来适应应用程序，或为应用程序选择不同的框架（或根本没有适用于应用程序的框架）。
- 了解一个框架要花很多时间和精力。（然而，学习从头开始开发类似的应用程序要花更多的精力。多个项目之后，学习框架所花的精力相比来说就算不了什么了。）
- 当你用框架时，你就放弃了应用程序设计的某些控制，通常包括程序的执行流。（这种控制的倒置意味着框架调用应用程序的特定对象。这就允许应用程序的开发者把精力集中于他所开发的对象的功能上，而忽略了它们与框架对象的合作。）

7.1.2 Java的AWT和Swing

到目前为止我们在本书中开发的交互程序都是基于文本的：用户在控制窗口输入文本，由 `ScanInput` 类来读入并解析。相反，如果用户使用基于GUI的程序，要做的事情就是用鼠标点击、拖动和选择、按按钮、选择菜单项、在文本区域输入等。目前广为使用的程序都支持图形用户界面。

Java的GUI框架由三个主要部分组成：组件、布局管理器和事件处理模型。组件是响应用户操作的可视化组件，如按钮、面板、对话框、菜单、文本区域和列表。组件在容器中出现，容器本身也是一种组件。这样就形成了一个包含层次结构，如3.4节一开始所示的图3-4中的例子一样。布局管理器安排容器中组件的位置。例如，组件可以沿着长方形排列，或像一段文字一样从左到右排列，或按指南针的五个区域排列（北，南，东，西，中）。Java的事件模型用来连接组件和事件处理行为。例如，当用户按一个按钮，观察按钮事件的对象（即所谓事件监听器）负责响应按按钮的动作。

在Java1.0中引进的AWT提供开发GUI的基本元素。AWT中的组件用对等组件（peer component）来实现，对等组件属于本地平台的自带的GUI系统。例如，当Java程序在Windows下面运行时，按钮用Windows系统下的按钮组件来实现，当在Macintosh下面运行时，按钮用Macintosh系统下的按钮组件来实现。这样程序看起来、用起来和它运行的环境保持风格上的一致。优点是用户一旦熟悉了某个平台，会认为Java程序和他熟悉的平台界面一致很舒服。AWT也有缺点，因为有些GUI系统比另外一些内容丰富，但它必须要按照大家都支持的部分来编程。如果要一个程序在所有系统下运行，只能用这些系统都支持的特性。即便当一个标准的组件在所有的GUI系统都支持，其行为也会因系统不同小有区别，这样程序员就很难实现跨平台的一致。因为这些原因，AWT相对简单，也就意味着缺少很多组件类型和现代GUI环境支持的特性。

这些问题在Java1.1和Swing中得到解决。Java1.1和更高版本支持创建所谓的轻型组件（lightweight component），即不依靠操作系统自带GUI系统的组件。（反之，AWT组件被称

为重型组件 (heavyweight component)。它们依赖于对等组件，常常要消耗很多系统资源。) 轻型组件完全由Java代码实现。组件功能由Java代码处理而不是依赖于平台自带的GUI系统时，程序在多个平台上的行为就可以保持一致。而且可能开发出在多个平台上风格一致，甚至可以由用户在运行时选择和改变的程序。

Swing是具有轻型组件特征的Java的用户界面程序库。Swing比AWT提供更大的组件集，而且Swing的组件通常来说更强大、特性更丰富。Swing没有取代AWT，而是建立在AWT的基础上。不过，在Swing和AWT提供相似服务的情况下（这种情况很常见），在本章中我们选择使用Swing。基于Swing的应用程序可以由Java 1.2或更高版本的解释器执行。不过支持Swing的浏览器广为使用只是一个时间的问题。

本章的其他部分探讨用于创建基于GUI程序的Java框架的组成单元。7.2节到7.4节包含创建基于GUI程序的三个组成部分：事件处理、组件和布局管理器。7.5 到7.7节用一个基于GUI的程序为例说明这些组成部分如何在一起工作，这个程序可以创建和编辑如多边形、矩形和椭圆这样的形状。

7.2 Java事件模型

这一节概述Java的事件模型，并采用一系列的程序来说明其用法。这组程序完成在平面上编辑点集和管理多边形。这里所讲的事件模型适用于Java 1.1或更高版本，也可用于Swing和AWT。我们将不讨论早于Java 1.0的事件模型，因为它几乎已经被废弃了。

7.2.1 概述

Java事件模型是基于观察者设计模式的。目标是根据用户的行动而产生事件的组件，观察者就是被通知有事件发生的客体，同时对事件进行响应。比如说，一个按钮可能是一个目标，它的观察者对按钮的动作进行响应。任何时候当用户按按钮时，观察者都获得通知。Java的事件模型看起来很复杂，这是因为它提供的大量的目标和事件类型，以及用于定制的不同技巧。不过透过它丰富的特点，只要记住Java的事件模型采用的是观察者模式即可。

在Java的事件模型下，目标被称为事件源 (event source) 而观察者被称为事件监听器 (event listener)。事件是由GUI组件根据用户的动作产生的。例如，当用户按按钮、输入文本到文本区域并按回车键或从菜单或列表中选择一项时，一个事件就产生了。GUI组件是事件源。当事件源产生一个事件后，它就通知每一个在事件源注册的事件监听器。事件源还提供方法以便注册对它的事件感兴趣的事件监听器。

事件源通过调用监听器的事件处理程序 (event handler) 之一来通知已注册的事件监听器有事件发生了。事件处理程序就是事件监听器对事件响应的方法。根据事件特性的不同，特定的事件处理程序被调用。(许多事件监听器定义多个事件处理程序)。一个包含和事件相关的信息的事件对象 (event object) 被传给事件处理程序，事件处理程序通过事件对象了解它必须响应的事件。每个事件对象都包含一个getSource方法来找出产生事件的组件 (事件源)。所有的事件对象都是java.util.EventObject类的子型。

可想而知，存在很多不同类型的事件，不同的事件源产生不同类型的信息。例如，当用户按一个按钮 (javax.swing.JButton类的实例) 时，按钮产生一个动作事件

(action event)。一个**ActionEvent**对象存储类似哪个按钮被按下了，按下按钮的同时哪个修改键（如Shift键或Control键）也被按下了这样的信息。另外一个例子是，当用户点击鼠标时，鼠标点击的组件就产生一个鼠标事件 (mouse event)。**MouseEvent**对象会取得以下信息：如鼠标的哪个键被按下；按下的位置相对于组件坐标空间的x和y坐标。本章的后面我们还会看到其他的例子。不过到目前为止，我们应该了解一个Java程序可能包含任何多个GUI组件充当事件源，每一个类型的事件源只能产生某些类型的事件。

讨论到现在，是应该考虑一个具体例子的时候了。假设我们正在开发一个应用程序，用来存储和在面板上显示的一组点。该程序有一个标有Triangulate的按钮，当按这个按钮时，面板创建及显示点集中的一组三角形（更多的三角形内容见练习6.18、图7-12程序界面）。我们假设包含点集的面板是用下面的语句创建的：

```
PointSetPanel panel = new PointSetPanel();
```

按钮Triangulate是用下面的语句创建的：

```
JButton triangulateButton = new JButton("Triangulate");
```

当一个事件源产生一个事件时，它通知已经注册的每一个事件监听器。事件源提供用于注册事件监听器的方法。在我们的例子中，panel在Triangulate按钮按下时有兴趣收到事件，这样panel就注册成一个由Triangulate按钮对象产生的动作的事件监听器。注册是由下面语句完成的：

```
triangulateButton.addActionListener(panel);
```

注册到某个事件源时，一个事件监听器暗示了它所感兴趣的事件类型。在这个例子中，panel表明用按钮的addActionListener方法注册而没有用按钮提供的其他注册方法来表明自己的兴趣。panel会在Triangulate按钮产生所有的动作事件时获得通知，而其他类型的事情产生时则不通知它。

为了注册一个事件源，事件监听器对象一定要是个正确的类型。在我们的例子中，panel一定可以监听动作事件，更确切地说，panel一定要实现接口**ActionListener**：

```
public interface java.awt.event.ActionListener {
    public abstract void actionPerformed(ActionEvent e);
}
```

panel实现这个接口的事实由按钮的addActionListener方法的签名来保证，用这个方法panel注册成为一个事件监听器：

```
// method of JButton class
public void addActionListener(ActionListener obj);
```

假如panel不是**ActionListener**接口的一个子型，它就不可能注册成为一个按钮动作事件的监听器（编译器将禁止这样做）。

当事件源产生一个事件时，它通知所有注册的监听器。特别地，事件源会调用每个已经注册的监听器的相应的事件处理方法，把包含事件信息的事件对象作为参数传给它。在我们运行的例子中，当用户按Triangulate按钮时，系统发给panel如下消息：

```
panel.actionPerformed(e)
```

参数e是描述按钮被按下的事件对象。这里panel一定要实现**actionPerformed**方法，

因为它实现了**ActionListener**接口。通常来说，当一个事件监听器被注册为一个事件源时，注册的动作就保证了事件监听器实现事件处理程序方法，通过这些方法事件源通知事件监听器有事件发生。

事件源同时提供方法来注销事件监听器。例如，按钮提供下面的方法用于注销监听器：

```
// method of JButton class
public void removeActionListener(ActionListener obj);
```

语句：

```
triangulateButton.removeActionListener(panel);
```

从感兴趣监听器的**triangulateButton**的集合中把**panel**删除；按钮的动作事件产生后，**panel**将不再被通知，直到它再次注册。

事件监听器的事件处理程序就是它响应事件的方法。在有动作事件的情况下，监听器保证只实现一个事件处理程序方法**actionPerformed**。（后面我们将看到有多个事件处理程序的事件监听）。在我们的例子里，**PointSetPanel**类的定义采用如下的形式：

```
public class PointSetPanel extends JPanel
    implements ActionListener {

    public void actionPerformed(ActionEvent e) {
        Object source = e.getSource();
        if (source == triangulateButton) {
            // respond to action events produced
            // by Triangulate button
            ...
        }
        // else handle action events generated
        // by other event sources
        // with which this panel is registered
        ...
    }

    // other methods
    ...
}
```

在**actionPerformed**方法的实现中，第一条语句得到指向事件源的引用。通常来说，当一个事件监听器同时监查多个事件源时，它必须分清楚现在所响应的事件源。在这种情况下，除了**triangulateButton**按钮以外，**panel**还可能监听其他的按钮，而**panel**的响应取决于哪个按钮被按下了。因为**getSource**方法是在**EventObject**类中定义的（该类是所有事件类型的父类），所以任何类型的事件都可以得到其事件源。

图7-1表明了本节的例子中类之间的关系的类图。**JButton**类从它的抽象父类**AbstractButton**继承了方法**addActionListener**和**removeActionListener**，以便于注册和注销事件监听器。**AbstractButton**类还定义了方法**fireActionPerformed**，每当按钮被按下时，该方法通知所有注册的监听器。注意图7-1符合观察者设计模式（参见图6-37）。

图7-2是显示我们例子中讨论的对象之间交互作用的顺序图。为了响应用户动作，按钮**triangulateButton**发给自己一个**fireActionPerformed**消息，这个消息反过来又导致一个**ActionPerformed**消息发送给注册的**panel**。顺序图表明**panel**注册到按钮中。然

而，事件监听器注册到事件源并不是必须的，尽管某些对象必须注册监听器才能得到事件通知。

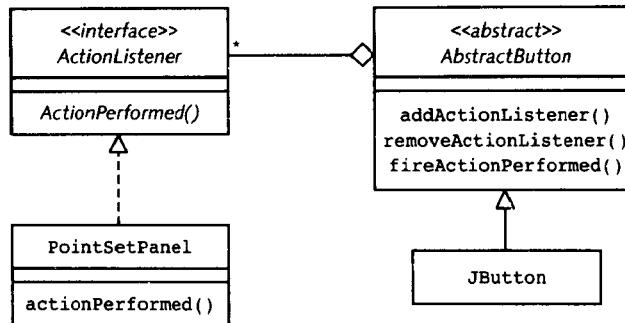


图7-1 基于观察者设计模式的Java的事件模型（比较图6-37）

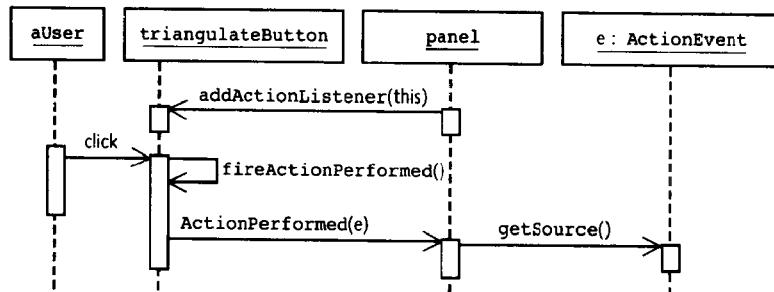


图7-2 顺序图，显示与Triangulate按钮一起注册面板，
以及后来当用户点击鼠标时响应

7.2.2 创建点集程序

从7.2.2节到7.2.5节，我们将开发四个基于GUI的程序，每个都建立在前一个的基础上。我们的第一个程序允许用户通过点击鼠标增加或删除点。第二个程序（7.2.3节）扩展了第一个程序，可以用鼠标的拖动来移动点。第三个程序是一个简单的多边形编辑器，允许用户创建、删除及通过鼠标的点击和拖动重定位多边形的顶点。最后一个程序（7.2.5节）引进了图形管理的接口，并用这个接口重新实现了7.2.3节的点编辑程序。设计这些程序的目的是介绍Java的事件模型。值得注意的是这些程序对Swing的依赖限制了面板的应用（我们将在7.3节探讨其他Swing组件的使用）。

我们的第一个基于GUI的程序是基本点集版本。点在框架中表现为彩色的小圆。用户可以在框架的背景上点击鼠标来创建出随机颜色的新点。要删除一个存在的点，用户只要点击这个点就可以删除它。

我们的clickPoints程序是一种面板，是Java的JPanel类的扩展。感兴趣的用户操作是点击鼠标。每当鼠标在ClickPoints面板上点击时，它就产生一个鼠标事件（mouse event），鼠标事件是java.awt.event.MouseEvent类的实例。在我们的实现中，面板监听它产生的每一个鼠标事件，即它把自己注册为自己鼠标事件的监听器。当事件处理逻辑相对简单时，这种观察产生的每个事件的方法是很有用的。下面是我们的clickPoints

类的结构：

```

public class ClickPoints extends JPanel
    implements MouseListener {

    // fields
    ...

    public static void main(String[] args) {
        ...
    }

    public ClickPoints() {
        // initialize this application
        ...
        // register this panel as a listener of
        // its own mouse events
        addMouseListener(this);
    }

    public void paintComponent(Graphics g) {
        // paint the current set of points into
        // the graphics context g
        ...
    }

    //
    // implement MouseListener interface
    // this program responds only to mouse clicks;
    // the remaining four event handlers of the
    // MouseListener interface do nothing
    //
    public void mouseClicked(MouseEvent e) {
        // handle mouse clicks
        ...
    }
    public void mouseEntered(MouseEvent e) { }
    public void mouseExited(MouseEvent e) { }
    public void mouseReleased(MouseEvent e) { }
    public void mousePressed(MouseEvent e) { }

    // additional helper methods
    ...
}

```

鼠标事件的监听器一定要实现`java.awt.event.MouseListener`接口。该接口规定了五个方法，用于响应一个组件的不同的鼠标动作。例如，当鼠标点击在一个组件上时，组件产生一个鼠标事件并调用每个监听器的`mouseClicked`方法，以便通知监听器该事件的发生。下面是这个接口的定义：

```

public interface java.awt.event.MouseListener {
    // invoked when the mouse is clicked on a component
    public void mouseClicked(MouseEvent e);

    // invoked when the mouse enters a component
    public void mouseEntered(MouseEvent e);

    // invoked when the mouse exits a component
    public void mouseExited(MouseEvent e);

    // invoked when a mouse button is pressed
}

```

```

    // on a component
    public void mousePressed(MouseEvent e);

    // invoked when a mouse button is released
    // on a component
    public void mouseReleased(MouseEvent e);
}

```

ClickPoints类实现了MouseListener接口，所以它必须实现这个接口所有的五个事件处理方法。而我们的应用程序只对点击鼠标产生的事件有兴趣，所以只有mouseClicked事件处理程序需要实现，其他四个事件处理程序什么也没做。我们的程序只响应鼠标点击。

下面来完成ClickPoints类的实现。至于其存储结构，我们将在一个称为figures的矢量中维护当前的点集。我们还会维护一个随机颜色生成器来产生点的颜色。

```

// fields of ClickPoints class
protected Vector figures;
protected RandomColor rnd;

```

下面的类定义包含公有接口的实现，不过只有保护型接口的说明：

```

public class ClickPoints extends JPanel
    implements MouseListener {

    protected Vector figures;
    protected RandomColor rnd;

    public static void main(String[] args) {
        JPanel panel = new ClickPoints();
        ApplicationFrame frame =
            new ApplicationFrame("ClickPoints");
        frame.getContentPane().add(panel);
        frame.show();
    }

    public ClickPoints() {
        setBackground(Color.black);
        figures = new Vector();
        rnd = new RandomColor();
        addMouseListener(this);
    }

    public void paintComponent(Graphics g) {
        super.paintComponent(g);
        Graphics2D g2 = (Graphics2D)g;
        g2.setRenderingHint(RenderingHints.KEY_ANTIALIASING,
                           RenderingHints.VALUE_ANTIALIAS_ON);
        Iterator iter = figures.iterator();
        while (iter.hasNext()) {
            Figure fig = (Figure)iter.next();
            fig.paint(g2);
        }
    }

    //
    // implement MouseListener interface
    //
    public void mouseClicked(MouseEvent e) {
        Figure clickedFig = findFigure(e.getX(), e.getY());
    }
}

```

```

        if (clickedFig == null)    // no figure was clicked
            addFigure(e.getX(), e.getY());
        else
            removeFigure(clickedFig);
        repaint();
    }

    public void mouseEntered(MouseEvent e) { }
    public void mouseExited(MouseEvent e) { }
    public void mouseReleased(MouseEvent e) { }
    public void mousePressed(MouseEvent e) { }

    protected Figure findFigure(int x, int y) {
        // EFFECTS: If (x,y) is contained in some point
        //   figure returns that figure; else returns null.
        ...
    }

    protected void addFigure(int x, int y) {
        // MODIFIES: figures, rnd
        // EFFECTS: Adds a new randomly colored point figure
        //   at position (x,y).
        ...
    }

    protected void removeFigure(Figure fig) {
        // MODIFIES: figures
        // EFFECTS: Removes the point figure fig.
        ...
    }
}

```

为了完成ClickPoints类的实现，需要实现它的三个保护型方法。这些用于增加和删除点的方法是很直接的。`addFigure`方法创建一个代表在 (x, y) 的新点，并且将新点加进矢量中。

```

// method of ClickPoints class
protected void addFigure(int x, int y) {
    PointGeometry point = new PointGeometry(x, y);
    Painter painter = new FillPainter(rnd.nextColor());
    figures.add(new Figure(point, painter));
}

```

`removeFigure`方法将`fig`从矢量`figures`中删除：

```

// method of ClickPoints class
protected void removeFigure(Figure fig) {
    figures.remove(fig);
}

```

剩下下一个保护型方法`findFigure`调用的时候传入整型参数 x 和 y ，返回与点 (x, y) 临近的点图形，如果没有这样的点图形存在就返回`null`。一个点 P 与点 (x, y) 相临近定义为两者之间的距离小于一个固定的距离 R 。可以想象，以点 (x, y) 为圆心，以 R 为半径为圆，测试矢量`figures`中的每一个点，直到找到点 P ， P 就被返回。如果没有找到符合条件的 P ，方法返回`null`。下面的实现假设 R 的值为3。

```

// method of ClickPoints class
protected Figure findFigure(int x, int y) {

```

```

EllipseGeometry disk =
    new EllipseGeometry(x-3, y-3, 6, 6);
Iterator iter = figures.iterator();
while (iter.hasNext()) {
    Figure fig = (Figure)iter.next();
    PointGeometry p = (PointGeometry)fig.getGeometry();
    if (disk.contains(p))
        return fig;
}
return null;
}

```

练习

- 设计一个功能与ClickPoints相似的程序clickAndColorPoints，不同的是当用户点击某个点时，不是把它删除，而是使其变色（颜色随机）。
- 设计一个功能与ClickPoints相似的程序clickEllipse，不同的是当用户点击框架背景时，产生一个椭圆，椭圆颜色随机，长和高的大小在[10..40]范围内随机取。
- 定义类PointZoneGeometry继承PointGeometry，包含方法contains判断输入点是否到已定义的点的距离小于R，R称为区域半径。测试时点区域图形就像一个圆盘，或是一个点。下面是类的框架说明，它只指出了不同于其从PointGeometry继承的方法的那些方法：

```

public class PointZoneGeometry
    extends PointGeometry
    implements AreaGeometry {
public PointZoneGeometry(int x, int y, int radius)
    throws IllegalArgumentException
// EFFECTS: If radius <= 0 throws
//   IllegalArgumentException; else constructs
//   a point at (x,y) and sets its zone radius
//   to radius.

public PointZoneGeometry(int x, int y)
// EFFECTS: Constructs a point at (x,y) and
//   sets its zone radius to 2.

public PointZoneGeometry(PointGeometry p,
    int radius)
throws NullPointerException,
    IllegalArgumentException
// EFFECTS: If p is null throws
//   NullPointerException; else if radius <= 0
//   throws IllegalArgumentException; else
//   constructs a point at p and sets its
//   zone radius to radius.

public PointZoneGeometry(PointGeometry p)
    throws NullPointerException
// EFFECTS: If p is null throws
//   NullPointerException; else constructs a
//   point at p and sets its zone radius to 2.

public PointZoneGeometry()
// EFFECTS: Constructs a point at the origin
//   and sets its zone radius to 2.

```

```

public boolean contains(int x, int y)
    // EFFECTS: Returns true if (x,y) lies within
    // zone radius units from this point;
    // else returns false.

public boolean contains(PointGeometry p)
    throws NullPointerException
    // EFFECTS: If p is null throws
    // NullPointerException; else if p lies
    // within zone radius units from this point
    // returns true; else returns false.

public void setZoneRadius(int newR)
    throws IllegalArgumentException
    // MODIFIES: this
    // EFFECTS: If newR <= 0 throws
    // IllegalArgumentException; else sets
    // the zone radius to newR.

public int getZoneRadius()
    // EFFECTS: Returns current zone radius.
}

```

例如，执行下面的一段代码：

```

PointZoneGeometry p = new PointZoneGeometry(5,10,4);
if (p.contains(new PointGeometry(5, 12)))
    System.out.println("print me");
if (p.contains(new PointGeometry(30, 40)))
    System.out.println("don't print me");

```

只输出“print me”，因为点(5, 12)到(5, 10)的距离小于4，而点(30, 40)到(5, 10)的距离大于4。

要实现contains方法，你可能要考虑创建一个椭圆形圆盘类，由它来完成查询点是否在某一区域内的工作。然后修改ClickPoints.findFigure方法，使它用点区域图形而不是用椭圆判断距离，然后运行ClickPoints程序。

7.2.3 编辑点集程序

我们下面将要设计的基于GUI的程序EditPoints是一个点集编辑程序。它是7.2.2节中ClickPoints的功能增强。在原有点的概念和程序功能的基础上，用户可以用鼠标拖动点移动到新的位置，点一直跟随着鼠标光标移动，直到用户释放鼠标按钮。

要实现这个程序，需要处理鼠标的各种事件，包括按下鼠标按钮、释放鼠标按钮和拖动鼠标等事件。我们已经知道，注册一个MouseListener事件监听器可以处理按下鼠标按钮和释放鼠标按钮事件。好在Java中提供了拖动鼠标事件接口MouseMotionListener，定义如下：

```

public interface java.awt.event.MouseMotionListener {
    // invoked repeatedly while the mouse is dragged
    // (with its button down) in a component
    public void mouseDragged(MouseEvent e);

    // invoked repeatedly while the mouse is moved
    // (with its button up) in a component
}

```

```
    public void mouseMoved(MouseEvent e);
}
```

作为鼠标事件源，为了注册和注销事件监听器，面板提供下面两个方法：

```
// methods of JPanel class
public void addMouseMotionListener(MouseMotionListener l)
public void
    removeMouseMotionListener(MouseMotionListener l)
```

面板按下面注册事件监听器aListerner：

```
panel.addMouseMotionListener(aListerner);
```

然后，当鼠标移动时，aListerner会收到这个鼠标事件的通知。特别地，当鼠标按钮没有按下不断移动鼠标时，系统会不断地发消息：

```
aListerner.mouseMoved(e)
```

其中e包含了鼠标事件的信息。类似地，当鼠标按钮按下并不断拖动鼠标时，系统会不断地发消息：

```
aListerner.mouseMoved(e)
```

总之，Java中有两种鼠标事件监听器，**MouseListener**（鼠标事件监听器）和**MouseMotionListener**（鼠标移动事件监听器）。**MouseListener**是在鼠标按钮变化或鼠标到达组件的边缘时收到事件；而**MouseMotionListener**是在鼠标移动过程收到事件。在程序**EditPoints**中，需处理两种类型的鼠标事件，所以需创建并注册两种监听对象到**EditPoints**面板组件中。这里**EditPoints**面板是鼠标事件源，但它并不观察它自己的事件，而是由注册的两个事件监听器对象负责处理。

事件监听器要正确响应鼠标事件，就必须访问**EditPoints**类的状态。我们是通过在**EditPoints**类中定义两个事件监听器类（内部类）来实现这种访问，这就允许它们直接使用**EditPoints**的域名字访问这些域。下面来看**EditPoints**类的实现：

```
public class EditPoints extends JPanel {

    protected Vector figures;
    protected RandomColor rnd;
    protected Figure clickedFig;
    protected PointGeometry clickedPoint;

    public static void main(String[] args) {
        JPanel panel = new EditPoints();
        ApplicationFrame frame =
            new ApplicationFrame("EditPoints");
        frame.getContentPane().add(panel);
        frame.show();
    }

    public EditPoints() {
        setBackground(Color.black);
        figures = new Vector();
        rnd = new RandomColor();
        addMouseListener(new EditPointsMouseListener());
        addMouseMotionListener(
            new EditPointsMouseMotionListener());
    }
}
```

```

}

public void paintComponent(Graphics g) {
    // same as method ClickPoints.paintComponent
    ...
}

// the following three methods are defined the
// same as their counterparts in class ClickPoints.
//

protected Figure findFigure(int x, int y) { ... }
protected void addFigure(int x, int y) { ... }
protected void removeFigure(Figure fig) { ... }

// inner class: mouse motion listener for mouse motion
//
class EditPointsMouseMotionListener
    extends MouseMotionAdapter {
    public void mouseDragged(MouseEvent e) {
        if (clickedFig != null) {
            clickedPoint.setX(e.getX());
            clickedPoint.setY(e.getY());
            repaint();
        }
    }
}

// inner class: mouse listener for mouse button actions
//
class EditPointsMouseListener extends MouseAdapter {
    public void mouseReleased(MouseEvent e) {
        if (clickedFig == null) // no figure was clicked
            addFigure(e.getX(), e.getY());
        else if (e.isControlDown())
            removeFigure(clickedFig);
        clickedFig = null;
        repaint();
    }

    public void mousePressed(MouseEvent e) {
        clickedFig = findFigure(e.getX(), e.getY());
        if (clickedFig != null)
            clickedPoint =
                (PointGeometry) clickedFig.getGeometry();
    }
}
}

```

每个内部类是由一个适配器类 (adapter class) 扩展而成的，例如 `EditPointsMouseListerner` 类扩展了 `MouseAdapter` 类，如图 7-3 所示。`MouseAdapter` 类实现了 `MouseListener` 接口提供的五个方法，但每个方法都是空操作：

```

public class java.awt.event.MouseAdapter
    implements MouseListener {
    public void mouseClicked(MouseEvent e) { }
    public void mouseEntered(MouseEvent e) { }
    public void mouseExited(MouseEvent e) { }
}

```

```

public void mousePressed(MouseEvent e) { }
public void mouseReleased(MouseEvent e) { }
}

```

因此，要定义一个鼠标事件监听器类，有两种方法可选：一是扩展MouseAdapter类，然后覆盖所需的方法。上面EditPointsMouseListener类就采用这种方法，扩展类MouseAdapter，然后覆盖MousePressed和MouseReleased方法，其他的方法不用管，因为MouseAdapter已经实现过了。另一种方法当然是直接实现MouseListener接口，这样必须把接口中的五个方法全部实现才可以，不管用不用这些方法。

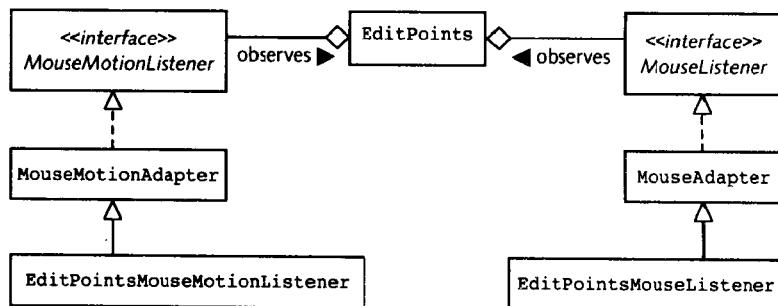


图7-3 EditPoints程序的结构

与此类似，Java提供了MouseMotionAdapter类，用来简化鼠标移动监听器的实现：

```

public class java.awt.event.MouseMotionAdapter
    implements MouseMotionListener {
    public void mouseDragged(MouseEvent e) { }
    public void mouseMoved(MouseEvent e) { }
}

```

我们已经知道，Java提供了两种不同的鼠标事件接口：MouseListener接口处理鼠标按钮事件和鼠标到达组件的边缘事件；MouseMotionListener处理鼠标移动事件。也许你要问，Java为什么要用两种接口处理？一个接口带上七个方法不是同样可以实现吗？答案是为了效率，由于鼠标的移动通常会很快，会频繁产生事件，然后进行处理，所以鼠标移动时会占用很多CPU资源。为了避免不必要的资源浪费，Java事件模型把这两种事件的处理分开。如果程序不需要处理鼠标移动事件，则只需使用MouseListener接口，这样可以提高程序的效率，7.2.2节中的ClickPoints程序就是这么实现的。当程序需要处理鼠标移动事件时，才使用MouseMotionListener接口，我们上面设计的EditPoints就属于这种情况。

练习

7.4 修改EditPoints程序的实现，使一个点被点击或被鼠标拖到一个新的位置时，如果与其他点有重迭，这个点显示在最上面（如它在任何其他点后输出）。

[提示：点的输出的顺序是由它们在figures矢量中的顺序决定]

7.5 写一个DragEllipses程序，它和练习7.2中的ClickEllipses有些相似，区别如下：用户可以用鼠标拖动椭圆移动位置，当用户在某一个椭圆上按下鼠标按钮并拖着移动时，整个椭圆图形随着鼠标移动，直到释放鼠标按钮，图形移动到释放位置处。

[提示：在`MousePressed`方法中，判断如果有图形被点击选中，则保存当前点击点位置；然后如果图形被鼠标拖动，在`MouseDragged`方法中按鼠标当前点位和鼠标前一个点位的差值移动椭圆，并保存当前点位。]

7.6 写一个`SweepRectangles`程序，它能在框架中使用鼠标拖动建立一组矩形：按下鼠标按钮时，一个长宽都为0的，位于鼠标点 (x_0, y_0) 的新的矩形建立（矩形内部填充颜色任意），随着移动鼠标到点 (x, y) ，不断变化的点 (x, y) 成为新的矩形点 (x_0, y_0) 的对角点，新矩形的建立直到释放鼠标点为止。当按着鼠标按钮不断移动鼠标时，要显示以前建立的所有图形和新建立的图形。

[提示：当按下鼠标时，保存点 (x_0, y_0) ，建立一个新的矩形图形，并把它加到图形集合中，随着鼠标的移动，修改当前矩形的属性并刷新所有的图形，你或许要建立下面保护型方法来刷新图形：

```
// method of SweepRectangles class
protected void updateCurrentRectangle(int x, int y)
```

其中 (x, y) 为当前鼠标位置， (x_0, y_0) 是最初点。注意， (x_0, y_0) 可能是矩形的四个角中的任一个角的点，这是由 x 和 y 值决定的。]

7.2.4 编辑多边形程序

在本节中，我们将设计一个用于编辑多边形的基于GUI的程序`EditPolygon`，它在面板上显示多边形并将其当前顶点高亮显示。当用户点击面板上某一点时，多边形创建一个新的顶点，并被插入当前顶点的后面，这个顶点变为当前顶点，并用高亮度表示；用户可以选中已有的点，并拖着它移动到一个新的位置，这个顶点变为当前顶点；用户可以选中顶点并按`Ctrl`键删除一个顶点，这个顶点的前一个顶点变成当前顶点。

`EditPolygon`程序和上一节的`EditPoints`程序比较相似，因为多边形的顶点可以看作是点集中的点。但它们的实现是有明显不同的。`EditPoints`类中定义了两个内部类作为鼠标事件监听器，而`EditPolygon`则定义独立的类`PolygonListener`监听处理鼠标事件并负责管理多边形图形。

`EditPolygon`类是从`JPanel`继承而来，它有两个`Figure`类型域：`polygonFigure`中保存多边形；`vertexFigure`保存当前顶点。除了定义一个构造器和一个`paintComponent`方法外，`EditPolygon`还定义了方法`setVertexPosition`，事件监听器利用它修改当前顶点的位置。`EditPolygon`类定义如下：

```
public class EditPolygon extends JPanel {

    protected Figure polygonFigure;
    protected Figure vertexFigure;
    final static PointGeometry InitialPoint =
        new PointGeometry(100, 100);

    public static void main(String[] args) {
        JPanel panel = new EditPolygon();
        ApplicationFrame frame =
            new ApplicationFrame("Edit Polygon");
        frame.getContentPane().add(panel);
        frame.show();
    }
}
```

```

public EditPolygon() {
    setBackground(Color.black);
    // create the polygon and vertex figures
    Painter fill = new FillPainter(Color.blue);
    Painter draw =
        new DrawDynamicPolygonPainter(Color.green);
    DynamicPolygonGeometry poly =
        new DynamicPolygonGeometry(InitialPoint);
    polygonFigure =
        new Figure(poly, new MultiPainter(fill, draw));
    vertexFigure =
        new Figure(InitialPoint, new FillPainter(Color.red));
    // create and register the listener
    PolygonListener listener =
        new PolygonListener(this, poly);
    addMouseListener(listener);
    addMouseMotionListener(listener);
}

public void paintComponent(Graphics g) {
    super.paintComponent(g);
    Graphics2D g2 = (Graphics2D)g;
    g2.setRenderingHint(RenderingHints.KEY_ANTIALIASING,
                        RenderingHints.VALUE_ANTIALIAS_ON);
    polygonFigure.paint(g2); // paint the polygon
    vertexFigure.paint(g2); // paint current vertex
}

public void setVertexPosition(PointGeometry p) {
    vertexFigure.setGeometry(p);
}
}
}

```

PolygonListener类是通过存储在它的iter域中的多边形迭代器来管理多边形的。因为EditPolygon类维护到迭代器基础多边形的引用，通过迭代器所做的对多边形的改变是自动与EditPolygon通信的。虽然如此，类似的变化发生时还是要通知EditPolygon；同时当顶点的位置发生变化时，也要通知EditPolygon。于是，PolygonListener保持对EditPolygon面板的引用，必要时通过它刷新图形。PolygonListener还定义了一个布尔型的域vertexBeingDragged，它的值用来标识是否有顶点在被拖动。PolygonListener类定义如下：

```

public class PolygonListener extends MouseAdapter
    implements MouseMotionListener {
    protected PolygonIterator iter;
    protected boolean vertexBeingDragged;
    protected EditPolygon panel;

    public PolygonListener(EditPolygon panel,
                          DynamicPolygonGeometry poly) {
        this.panel = panel;
        iter = poly.iterator();
        panel.repaint();
    }

    public void mousePressed(MouseEvent e) {
        vertexBeingDragged = findVertex(e.getX(), e.getY());
    }
}

```

```

public void mouseReleased(MouseEvent e) {
    if (vertexBeingDragged && e.isControlDown())
        removeVertex();
    else if (!vertexBeingDragged) {
        PointGeometry p =
            new PointGeometry(e.getX(), e.getY());
        insertNewVertex(p);
    }
    vertexBeingDragged = false;
    panel.repaint();
}

public void mouseDragged(MouseEvent e) {
    if (vertexBeingDragged) {
        moveVertex(e.getX(), e.getY());
        panel.repaint();
    }
}

public void mouseMoved(MouseEvent e) { }

// 
// protected interface
//
protected boolean findVertex(int x, int y) {
    PointZoneGeometry disk = new PointZoneGeometry(x, y);
    for (int i=0;i<iter.nbrVertices(); iter.next(),i++) {
        PointGeometry p = iter.point();
        if (disk.contains(p)) {
            panel.setVertexPosition(p);
            return true;
        }
    }
    return false;
}

protected void insertNewVertex(PointGeometry p) {
    iter.insertAfter(p);
    panel.setVertexPosition(p);
}

protected void removeVertex() {
    if (iter.nbrVertices() > 1) {
        iter.remove();
        panel.setVertexPosition(iter.point());
    }
}

protected void moveVertex(int x, int y) {
    iter.moveTo(x, y);
    panel.setVertexPosition(new PointGeometry(x, y));
}
}

```

7.2.5 重设计编辑点集程序

从5.6节开始接触Figure类开始，我们都有意区分几何图形和它的外观的区别。几何图形是要实现接口Geometry的，而它的外观是要实现Painter接口的。Figure对象是包含这两个部分。同样的道理，理解区分涉及几何图形（geometry）的操作和涉及图形

(`figure`) 的操作也是很重要的。但在 7.2.3 节中的 `EditPoints` 程序中却几乎看不到这些区别。当用户点击鼠标创建一个新点时, 这是必要的几何图形操作步骤, 鼠标用于指出平面中的位置, `EditPoints` 程序为了响应事件, 它创建了一个点几何图形, 然后创建了一个任意颜色的绘图工具 (painter), 由它们组成一个新的图形。

因此在这一节中, 我们重新设计 `EditPoints` 程序, 来看如何有意识地区分这三个部分的操作。这样做要引进一个新的接口 `FigureManager`, 称为图形管理器, 它主要描述管理图形和外观所需的操作。设计图形管理器的主要目的是使客户能够摆脱封装图形中的几何图形和管理图形的责任。由于我们所设计的 `EditPoints` 程序很简单, 所以你也许会发现图形管理器用在下面的程序中, 和前面的程序相比并没有简化多少。但在后面更复杂的程序中, 它的优点就会表现出来。

从用户角度来看, 新的程序 `EditPointSet` 和 `EditPoints` 一样, 但从设计的角度来看则不同 (参见类图 7-4)。`EditPointSet` 中定义了一个 `EditPointSetManager` 类的实例——图形管理器。`EditPointSet` 通过发消息给它的图形管理器来管理图形。`EditPointSet` 自己管理几何图形: 创建、删除、移动点等。同 `EditPoints` 类一样, `EditPointSet` 类是定义了两个鼠标事件监听器作为内部类的面板。

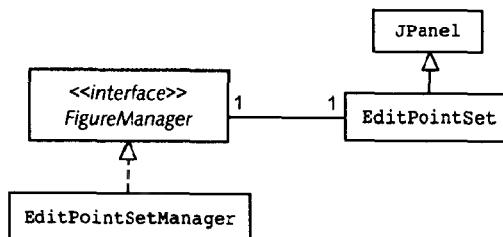


图 7-4 `EditPointSet` 程序的结构

先来看 `FigureManager` 接口。`FigureManager` 接口指定封装图形中的几何图形的行为 (特别是分配外观给几何图形) 以及管理图形集合的行为。在任何时候, 最多有一个选中的几何图形 (`selected geometry`) (对于大多数应用而言, 维护那些适用于选中几何图形的操作是很有帮助的。例如, 大多数画图程序支持选中几何图形的概念, 对选中几何图形可以被变换、编辑、赋予新的外观等操作)。`FigureManager` 接口中 `Select` 操作从图形集中选择一个几何图形, 而 `selected` 操作则返回当前选中的几何图形; `add` 操作把一个传入的几何图形对象转成图形后加入到图形集中; `remove` 方法负责把传入的几何图形的图形从图形对象集中删除; `size` 方法返回图形集合中的几何图形数; `get` 按下标值访问几何图形。也可以使用 `find` 操作访问几何图形: 以某一个点 p 的 x 轴和 y 轴坐标调用时, `find` 返回由 p 点 “命中” 的某个集合中的几何图形。

无论何时图形管理器以明显的方式发生变化, 它的客户都通过发送 `updateManager` 消息来通知它。图形管理器还提供了 `getFigures` 方法返回一个表示它的图形集的节点; 这样客户可以用下面语句画出这一图形:

```
aFigureManager.getFigures().paint(g2);
```

其中 `g2` 是绘图环境。`FigureManager` 接口的定义如下:

```
public interface FigureManager {
```

```

public void select(Geometry g)
    throws IllegalArgumentException;
// MODIFIES: this
// EFFECTS: If g is null deselects the selected
// geometry (if any); else if g belongs to
// this collection selects g; else throws
// IllegalArgumentException.

public Geometry selected();
// EFFECTS: Returns the selected geometry if any;
// else returns null.

public Node getFigures();
// EFFECTS: Returns a node representing the set
// of figures.

public void updateManager();
// MODIFIES: this
// EFFECTS: Any.

public void add(Geometry g);
// MODIFIES: this
// EFFECTS: Wraps g in a figure and adds it to
// its collection.

public void remove(Geometry g);
// MODIFIES: this
// EFFECTS: If g belongs to some figure in
// the collection, removes the figure;
// else does nothing.

public Geometry get(int i)
    throws IndexOutOfBoundsException;
// EFFECTS: If 0 <= i < size() returns the i'th
// geometry; else throws IndexOutOfBoundsException.

public int size();
// EFFECTS: Returns the number of geometries
// in this collection.

public Geometry find(int x, int y);
// EFFECTS: Returns some geometry in this collection
// hit by the point (x,y); returns null if no
// such geometry exists.
}

```

FigureManager接口中方法的后置条件比较弱，需要由实现接口的类来加强。比如**find**方法中“点p命中一个几何图形”可以理解为点p包含在几何图形中或点p接近几何图形或其他意思。再如**updateManager**方法的影响没有任何描述，这就是说图形管理器可以依据程序要求的方式更新。

EditPointSet类的实现可以作为如何利用图形管理器的例子，下面是它的类定义：

```

public class EditPointSet extends JPanel {

    protected FigureManager manager;
    protected PointGeometry clickedPoint;
    public static void main(String[] args) {
        JPanel panel = new EditPointSet();
        ApplicationFrame frame =

```

```

        new ApplicationFrame("EditPointSet");
        frame.getContentPane().add(panel);
        frame.show();
    }

    public EditPointSet() {
        setBackground(Color.black);
        makeFigureManager(this);
        addMouseListener(new EditPointSetMouseListener());
        // the following mouse motion listener is
        // implemented as an anonymous inner class
        // (see the explanation below)
        addMouseMotionListener(new MouseMotionAdapter() {
            public void mouseDragged(MouseEvent e) {
                if (clickedPoint != null) {
                    clickedPoint.setX(e.getX());
                    clickedPoint.setY(e.getY());
                    manager.updateManager();
                }
            }
        });
    }

    public void makeFigureManager(JPanel canvas) {
        manager = new EditPointSetManager(canvas);
    }

    public void paintComponent(Graphics g) {
        super.paintComponent(g);
        Graphics2D g2 = (Graphics2D)g;
        g2.setRenderingHint(RenderingHints.KEY_ANTIALIASING,
                            RenderingHints.VALUE_ANTIALIAS_ON);
        manager.getFigures().paint(g2);
    }

    class EditPointSetMouseListener extends MouseAdapter {
        public void mouseReleased(MouseEvent e) {
            if (clickedPoint == null) // no figure was clicked
                manager.add(new PointGeometry(e.getX(), e.getY()));
            else if (e.isControlDown())
                manager.remove(clickedPoint);
            clickedPoint = null;
            manager.updateManager();
        }

        public void mousePressed(MouseEvent e) {
            clickedPoint =
                (PointGeometry)manager.find(e.getX(), e.getY());
            manager.updateManager();
        }
    }
}

```

EditPointSet同EditPoints的区别在于图形管理方法的不同。EditPoints要自己管理图形，所以它要有保存图形的域并有相应的处理图形的保护型方法。而EditPointSet则由保存在域manager中的图形管理器负责处理。例如，mouseReleased方法中需要增加和删除图形操作，这是通过向图形管理器发送add和remove消息完成的。

在EditPointSet的构造器中，鼠标移动事件监听器实现为一个匿名内部类(anonymous inner class)，监听器是方法addMouseMotionListener的参数。匿名类扩

展MouseMotionAdapter类，除了覆盖mouseDragged方法外，其他的方法不用管。匿名内部类在只需要类的一个实例的情况下时很有用的。如果事件监听器的定义很简单，并且只需要它的一个实例，通常我们会把它定义为匿名内部类。

接着考虑类EditPointSetManager的实现。它定义了三个实例域：canvas域保存画图面板的引用，更新图形管理器时要发送paint消息给面板；node保存包含点集的组节点；rnd中保存一个随机颜色生成器，用它产生的颜色画点。下面是EditPointSetManager的类定义：

```

public class EditPointSetManager
    implements FigureManager {

    protected JPanel canvas;
    protected GroupNode node;
    protected RandomColor rnd;

    public EditPointSetManager(JPanel canvas) {
        this.canvas = canvas;
        node = new GroupNode();
        rnd = new RandomColor();
    }

    public Node getFigures() {
        return node;
    }

    public void updateManager() {
        canvas.repaint();
    }

    public void add(Geometry p) {
        Painter painter = new FillPainter(rnd.nextColor());
        Figure fig = new Figure(p, painter);
        node.addChild(fig);
    }

    public void remove(Geometry p) {
        for (int i = 0; i < node.nbrChildren(); i++) {
            Figure fig = (Figure)node.child(i);
            Geometry g = fig.getGeometry();
            if (g == p) {
                node.removeChild(i);
                return;
            }
        }
    }

    public Geometry find(int x, int y) {
        PointZoneGeometry disk = new PointZoneGeometry(x,y);
        for (int i = node.nbrChildren()-1; i >= 0; i--) {
            Figure fig = (Figure)node.child(i);
            PointGeometry point =
                (PointGeometry)fig.getGeometry();
            if (disk.contains(point))
                return point;
        }
        return null;
    }

    public Geometry get(int i)
}

```

```

        throws IndexOutOfBoundsException {
    Figure fig = (Figure)node.child(i);
    return fig.getGeometry();
}

public int size() {
    return node.nbrChildren();
}

// methods of FigureManager interface
// not used by this application
public void select(Geometry g)
    throws IllegalArgumentException { }
public Geometry selected() { return null; }
}

```

本节中主要讲述了基于观测者模式的Java事件模型。事件源（通常为GUI组件）是目标，事件监听器是观测者。当事件源触发事件时，通过调用每个注册的事件监听器的事件处理程序通知事件发生，同时通过事件处理程序传入事件对象，由事件处理程序完成相应处理。

如果要按要求定制处理事件方法的行为，那么就需定义事件监听器类，在这个类中实现事件源所期望的接口。举例来说，如果事件监听器希望处理按钮所产生的事件，那么事件监听器类就要实现**ActionListener**接口。这是Java中组合定制的例子（黑盒定制），因为事件监听器是组件，框架依据需要来使用它们。

要定制事件源，可以扩展框架提供的GUI组件类，即扩展类可以定义新的行为并覆盖原有行为。例如，**EditPointSet**通过扩展Java的**JPanel**类来定义，它代表一个专用的面板，它覆盖了**paintComponent**方法并增加了新的行为。这是Java中继承定制的例子（白盒定制）。

练习

7.7 修改类**EditPointSet**的实现，把原来由**EditPointSetMouseListener**实现的鼠标事件监听器类用一个匿名内部类来实现。

7.8 编写程序**EditRectangles**，它和练习7.6的**SweepRectangles**相似，但有两点不同：首先，只有在框架的背景上开始拖动鼠标时，才能创建新的矩形；如果在一个已存在的矩形内部开始拖动，则不产生新的矩形。其次，按**Ctrl**键并点击图形则删除它（如果所点击位置上有多个矩形，只删除一个）。利用图形管理器，应该定义类**EditRectanglesManager**实现**FigureManager**接口，**EditRectangles**使用图形管理器**EditRectanglesManager**提供的服务。

7.3 组件

图形用户接口可能包括按钮、面板、标签、文本域、滚动条、下拉列表框和其他组件。相关的组件包括在容器中。容器可包括组件和其他容器，这样一个容器和组成它的组件的层次关系称为包含层次结构（见前面图3-4）。包含层次结构的根节点是图形程序的顶层窗口框架（frame），叶节点是原子组件，其他中间节点为容器。

在Java提供的组件工具箱AWT中，所有的组件类都是抽象类**java.awt.Component**类的子型，其中的一个子类**java.awt.Container**表示容器。组件类和容器类之间是组

合设计模式关系：Container是Component类的子型，而Container对象又是由零个或多个Component子型的实例组成。

Swing是Java中基于AWT之上的另一个图形分支。Swing定义了抽象类javax.swing.JComponent，扩展了Container类。Swing组件类都是JComponent的子型，它们在Java中是已经完全定义过的类，可以直接使用。图7-5中显示了本章中所讨论的GUI组件，这个图中仅描述了AWT和Swing中的类和接口的一小部分。

按照约定，Swing组件类名以字母J开始。多数情况下，Swing组件类名在相对应的AWT组件类名前加字母J。举例来说，如在AWT中按钮类为Button，则在Swing中为JButton。另外，Swing中还定义了很多组件类型，这些组件类型在AWT中却没有相对应的部分。

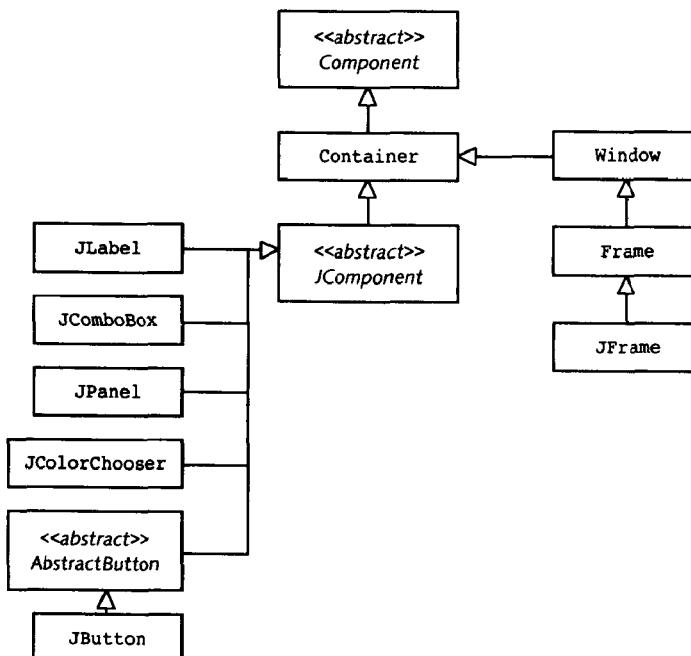


图7-5 本章中用到的GUI组件图

7.3.1 Component和Container类

除了AWT中和菜单相关的组件外，所有的GUI组件都以抽象类java.awt.Component为父类。该类提供了两个与颜色相关的特性：foreground（前景）和background（背景），它们在不同的组件中有不同的作用。为响应用户的鼠标点击和鼠标移动动作，组件需要能产生鼠标事件，Component类定义了以下注册和注销事件监听器的方法：

```

public void addMouseListener(MouseListener l);
public void removeMouseListener(MouseListener l);
public void
addMouseMotionListener(MouseMotionListener l);
public void
removeMouseMotionListener(MouseMotionListener l);
  
```

java.awt.Container类用于建立容纳组件的容器对象。该类定义了多个add与

`remove`的重载方法，使容器可以以不同的方式增加和删除组件。以增加组件为例，一种`add`方式是在容器的最后增加一个组件；另一种是在位置`indx`插入增加组件（假定第一个组件位置为0）：

```
public void add(Component component);
public void add(Component component, int indx);
```

同样`Container`类中定义了按组件名和按组件的位置删除组件的`remove`方法：

```
public void remove(Component component);
public void remove(int indx);
```

`getComponents`方法返回容器中所有的组件数组，`getComponent`方法返回某一位置的组件：

```
public Component[] getComponents();
public Component getComponent(int indx);
```

`Container`类中还有一个其值为`LayoutManager`对象的特性`layout`，特性的值确定容器中的组件如何在输出时布局排列。关于布局管理器将在7.4节中介绍。

7.3.2 JComponent类

`javax.swing.JComponent`抽象类是整个Swing组件层次结构图的根，也就是说除了顶层的容器如框架之外，所有的Swing组件都是`JComponent`的子型。除了继承父类`Container`的行为外，`JComponent`类又提供了很多有用的行为。其中`setToolTipText`方法用来设定光标停在组件上时弹出的提示信息；该类定义了`border`特性，其值用来表示边框（边框有两种：一种是可修饰的，如凸起、蚀刻、带标题等；另一种是不可修饰的，如填料、简单表单组合）；`JComponent`类提供一组与布局管理器相关的方法`setPreferredSize`、`setMinimumSize`和`setMaximumSize`，用来调整组件在容器中的大小，分别表示设置组件的最佳、最小和最大尺寸；同时`JComponent`使用双缓冲输出框架序列（frame sequence）（组件被绘在幕后缓冲区，它一次全部显示）效果更好，除了上面提到的，`JComponent`还提供了许多其他的服务。

7.3.3 JPanel类

`JPanel`类是Swing中组件的一个常用容器称为面板。一个面板在程序中输出时是按背景、边框、组件的顺序输出的。如果要增加一个组件到面板中，需要发送`add`消息给面板，输入参数为组件对象。例如，下面语句增加一个按钮到面板`aPanel`中：

```
aPanel.add(new JButton("Press me"));
```

`add`是从父型`Container`继承的重载方法。加入面板容器的组件的排列位置是由它的布局管理器确定的。

组件不能直接放到顶层的窗口（`JFrame`）中。首先调用`JFrame`的`getContentPane`获得框架的内容格，它是保存框架组件的容器，然后调用容器的`add`方法将`JPanel`加到内容格中，最后再把组件（包括嵌套容器）加入到面板中：

```
JPanel topPanel = new JPanel();
frame.getContentPane().add(topPanel);
```

```
// build and add new components to topPanel
...
```

另一种实现方法是创建一个新 JPanel 使它成为框架的容器格：

```
JPanel topPanel = new JPanel();
frame.setContentPane(topPanel);
// build and add new components to topPanel
...
```

7.3.4 JButton类

JButton类用于实现按钮，图7-6中对话框下面的三个按钮就是 JButton类实例。一个按钮上可显示文字或图标，它与 JButton类的特性text和icon相对应。这两个特性值可以由类的构造器初始化，下面为四个构造器：

```
public JButton(String text, Icon icon);
public JButton(String text);
public JButton(Icon icon);
public JButton();
```

除了通过构造器初始化之外，text和icon特性分别可以用方法setText和setIcon设定。

当用户点击按钮后，按钮触发一个动作事件，通过actionPerformed方法每个事件监听器（ActionListener）都会收到事件通知，ActionListener接口的定义如下：

```
public interface java.awt.event.ActionListener {
    public void actionPerformed(ActionEvent e);
}
```

当然， JButton类中同样提供注册和注销事件监听器的方法addActionListener和removeActionListener。这两个方法和 JButton类的大部分行为一样，都是在它的父类AbstractButton中实现的。

7.3.5 JLabel类

JLabel类用于显示静态的文字或图像，静态的意思是文字输出后不允许修改（允许修改的文字是由类JTextComponent生成）。JLabel类有特性text和icon，其值被标签显示。文字的字体是由从Component类继承的特性font决定。JLabel类还提供了调整输出内容和标签边界相对位置的方法。标签常放在同一容器其他组件的边上，用来说明其他组件的用途。例如，常常在文本框或组合框边上加一个标签说明框的用意，因为它们本身没有说明标签。

7.3.6 JComboBox类

组合框也称为下拉选择项列表，它包括两种形式：一是可修改的，另一种是不可修改的。在可修改形式中，用户可以修改选择的项的值并可以输入新的值（已有的选择列表不受影响）。默认情况下，组合框是不可修改的，这就暗示用户只能从列表中出现的项中选择。图7-11和7-12中的程序界面都包含一个组合框，它们为右侧带有一向下三角形矩形框，左侧为当前选择项。图7-11中还可以看到下拉列表的选项是一组颜色图标，点选其中之一就选择一种新的颜色。

组合框中的选择列表项可以在创建对象时输入，下面是为此提供的两个构造器：

```
public JComboBox(Object[] items);
public JComboBox(Vector items);
```

下面语句创建有四种颜色选项的组合框：

```
String[] colorNames = {"red", "green", "blue", "purple"};
JComboBox colorBox = new JComboBox(colorNames);
```

JComboBox还提供了一个没有参数的构造器，用它创建没有任何初始项的组合框。

项从0开始按下标排列，JComboBox提供多个方法用来确定选择项。其中getSelectedItem方法返回选择项，getSelectedIndex返回下标值：

```
public Object getSelectedItem();
public int getSelectedIndex();
```

对可修改组合框，如果没有选择下拉列表项，则getSelectedIndex方法会返回-1。因为JComboBox有两个特性selectedItem和selectedIndex，因此下面两个方法可用来选择项：

```
public void setSelectedItem(Object obj);
public void setSelectedIndex(int idx);
```

如果setSelectedItem输入参数obj不在选择项列表中，则调用它后会选择列表中的第一个项。一般情况下，项是由用户选择决定的而不是程序决定的。

选择列表是可以动态增加和删除，这要由方法addItem和removeItem完成：

```
public void addItem(Object obj);
public void removeItem(Object obj);
```

addItem在选择列表后追加一个项obj，removeItem删除obj项。当然JComboBox还提供了以下下标定位增加和删除项的方法。

用户选择改变时，组合框产生一个动作事件，注册的事件监听器会收到这个事件。JComboBox类提供addActionListener和removeActionListener方法来注册和注销事件监听器。注册的事件监听器常常定义如下形式的方法actionPerformed：

```
public void actionPerformed(ActionEvent e) {
    JComboBox source = (JComboBox)e.getSource();
    String item = (String)source.getSelectedItem();
    // process based on the selection item
    ...
}
```

有时组合框不需注册事件监听器，这种情况下，组合框选项只表示一个状态变量，程序只需通过查询当前选择就可知当前状态值。

7.3.7 JColorChooser类

如图7-6所示，JColorChooser提供一个用户可以选择颜色的对话框。对话框由两个部分组成：上面的选择面板（chooser panel）和下面的预览面板（preview panel）。预览面板显示所选择颜色的效果，选择面板供用户使用三种方法之一选择一种颜色。三种方法分别用三张带有趣头的卡片表示：（a）题头为Swatch表示从颜色样本表中选择；（b）题

头为HSB表示依颜色的色彩饱和亮度（hue-saturation-brightness）选择；（c）题头为RGB表示依红绿蓝（red-green-blue）颜色值选择。如果对象框是有模式的（即用户在继续进行之前必须响应对话框的请求），需包括三个按钮：OK确认选择并关闭对象框；Cancel放弃选择并关闭对话框；Reset恢复对话框初始状态。

JColorChooser有三种使用方法。其中最简单的是调用下面的静态方法显示有模式对话框：

```
public static Color showDialog(Component parent,
                               String title,
                               Color initialColor)
```

parent是对话框的父组件（如果对话框不需要显示在父框架的最前面，则此参数为null），title是对话框的题头，initialColor是对话框初始值。当用户关闭对话框时，showDialog返回值与用户操作有关：如果用户按OK按钮，返回所选择颜色；如果用户按Cancel按钮或点击对话框关闭按钮，则返回null。在本书中用这种方式使用JColorChooser，如图7-6所示。

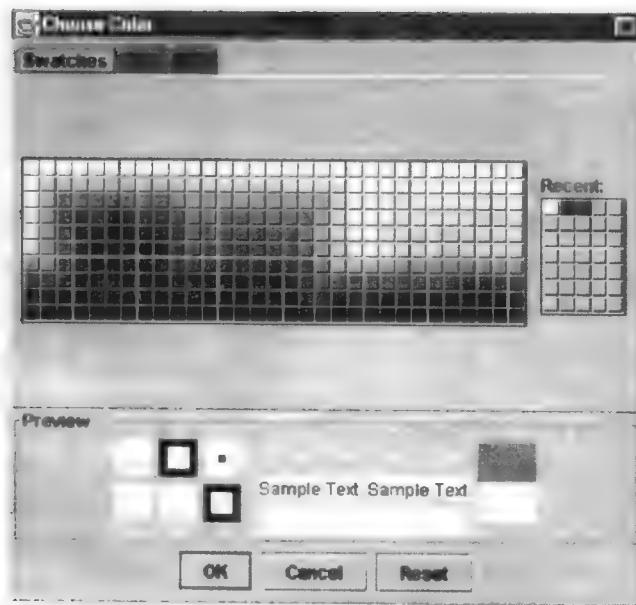


图7-6 颜色选择对话框

普遍使用的第二种方法是用JColorChooser类的静态方法createDialog创建新的对话框。使用这种方法用户可以选择创建无模式或有模式对话框，带有OK和Cancel按钮和相应的消息响应。使用这个类的最后一种方法是把JColorChooser类的实例当作组件加入到任何容器中，它的color特性变化时颜色选择器注册的特性变化监听器会得到通知。

7.4 布局管理器

布局管理器（layout manager）用来控制组件在容器中的排列方式。`java.awt.LayoutManager`接口定义了布局管理器的一般行为。如图7-7所示，Java定义了一些`LayoutManager`接口的实现。分配给一个容器的布局管理器决定容器中组件的排列位置。

向容器发送setLayout消息来为它分配一个布局管理器：

```
aContainer.setLayout(aLayoutManager);
```

除了我们将要讨论的流式布局、网格布局、边界布局外，AWT和Swing中定义了很多其他类型的布局，其中网格包布局（`GridBagLayout`）是最灵活强大的，也是最不好使用的。

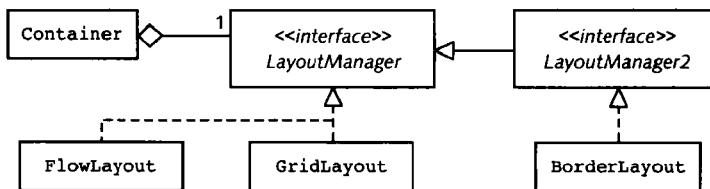


图7-7 本章中使用的布局管理器

7.4.1 流式布局

流式布局（`flow layout`）是把容器中的组件按从左到右、从上到下的顺序一行一行地排列，如同字处理器中的文字段一行一行的显示。组件在容器中先后位置是与它们加入的先后相对应。下面一段程序产生图7-8的框架：

```

public class TryFlowLayout {
    public static void main(String[] args) {
        ApplicationFrame frame =
            new ApplicationFrame("Flow Layout");
        Container pane = frame.getContentPane();
        pane.setLayout(new FlowLayout(FlowLayout.LEFT));
        pane.add(new JButton("Button one"));
        pane.add(new JButton("Button two"));
        pane.add(new JButton("Button three"));
        pane.add(new JButton("Button four"));
        frame.show();
    }
}
  
```

`TryFlowLayout`程序中按钮没有注册事件监听器，所以点击按钮时没有反应。每个排列行的按钮是居左、居中还是居右与传入`FlowLayout`构造器的参数有关（没有参数时为默认居中）。`FlowLayout`类定义了静态的参数`LEFT`、`CENTER`、`RIGHT`分别表示居左、居中和居右。同大多数布局管理器一样，当容器的大小改变时，组件的位置会随着重组。图7-8表明两个不同大小的框架的按钮的分布。



图7-8 流式布局管理器

7.4.2 网格布局

利用网格布局 (grid layout) 可以创建指定行数和列数的等大小的矩形单元网格，每个组件填满一个格子。当容器大小改变时，网格大小跟着变化，组件也随着变化。GridLayout类的构造器带有两个正整数参数，分别为网格的行和列。图7-9是由下面程序段放置2行3列网格的六个按钮产生的：

```
public class TryGridLayout {
    public static void main(String[] args) {
        ApplicationFrame frame =
            new ApplicationFrame("Grid Layout");
        Container pane = frame.getContentPane();
        pane.setLayout(new GridLayout(2, 3));
        pane.add(new JButton("Button one"));
        pane.add(new JButton("Button two"));
        pane.add(new JButton("Button three"));
        pane.add(new JButton("Button four"));
        pane.add(new JButton("Button five"));
        pane.add(new JButton("Button six"));
        frame.show();
    }
}
```

GridLayout类还提供了一个四个参数的构造器如下：

```
public GridLayout(int rows, int cols, int hgap, int vgap);
```

后面两个参数用于在组件之间增加间隔，hgap表示相邻列之间的水平间隔（以像素为单位），vgap表示相邻行之间的垂直间隔。两个参数的构造器则取后两个参数为0。



图7-9 网格布局

7.4.3 边界布局

边界布局 (border layout) 把容器分成五个区域：北、南、东、西、中，至多有五个组件放在这个容器中，每个区域一个组件。中间区域扩展为足够大，而其他四个则能够容纳它的组件即可。当增加一个组件到容器中时，要指明放置的区域，例如，下面的语句把aComponent放到aContainer的南部区域：

```
aContainer.add(aComponent, BorderLayout.SOUTH)
```

BorderLayout类提供了五个静态常量NORTH、SOUTH、EAST、WEST和CENTER分别表示五个对应区域。图7-10的框架是由下面程序段产生的：

```
public class TryBorderLayout {
    public static void main(String[] args) {
        ApplicationFrame frame =
```

```

    new ApplicationFrame("Border Layout");
    Container pane = frame.getContentPane();
    pane.setLayout(new BorderLayout());
    pane.add(new JButton("North"), BorderLayout.NORTH);
    pane.add(new JButton("South"), BorderLayout.SOUTH);
    pane.add(new JButton("East"), BorderLayout.EAST);
    pane.add(new JButton("West"), BorderLayout.WEST);
    pane.add(new JButton("Center"), BorderLayout.CENTER);
    frame.show();
}
}

```

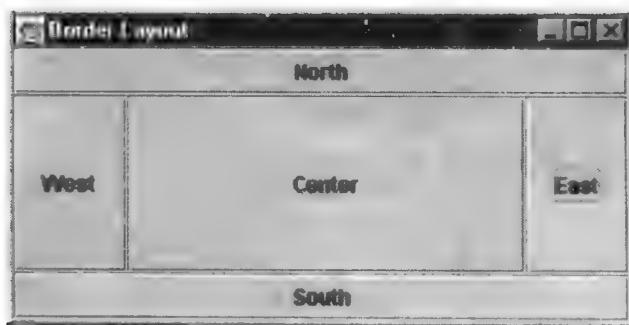


图7-10 边界布局

由于组件的位置由add方法的第二个参数决定，所以组件加入的次序是无关紧要的。BorderLayout类还提供了一个带有两个参数的构造器用来加大组件之间的距离：

```

public BorderLayout(int hgap, int vgap);

```

hgap以像素表示横向间距，vgap以像素表示纵向间距。

7.5 组件和事件监听器

本节中将利用两个简单的显示不同颜色的面板程序，示例说明如何把组件和事件监听器组合在一起使用。第一个程序让用户点击按钮选择颜色，而第二个程序中又增加了从组合框中选择并能记录用户的选择到组合框选项，供将来选择使用。

7.5.1 处理颜色

利用边界布局管理器，ColorPlay程序将框架分成两个面板：占框架大部分的中间面板显示当前颜色，南部区域面板包括四个按钮。程序界面与图7-11相似，只是没有右下角的组合框。点击选择red按钮，上边面板变成红色，green变绿，blue变蓝。选择custom时，弹出一个颜色选择对话框，用户选择其中一种颜色并按确定关闭返回，上边面板变成相应颜色；如果用户选择取消返回，则颜色不改变。

下面的ColorPlay类的实现创建了一个事件监听器对象供四个按钮公用。当监听器的事件处理程序actionPerformed被调用时，它获得对事件源（四个按钮之一）的引用，然后分别处理。下面是完整的程序：

```

public class ColorPlay extends ApplicationFrame {
    protected JPanel canvas, controls; // two main panels

```

```
protected JButton redButton, greenButton,
                blueButton, customButton;
protected Color color; // the current color

public static void main(String[] args) {
    JFrame frame = new ColorPlay("Color Play");
    frame.show();
}

public ColorPlay(String title) {
    super(title);
    Container topPane = getContentPane();
    topPane.setLayout(new BorderLayout());
    // creates and adds the canvas panel to the frame
    canvas = new JPanel();
    topPane.add(canvas, "Center");
    // creates and adds the control panel to the frame
    controls = new JPanel();
    controls.add(redButton = new JButton("red"));
    controls.add(greenButton = new JButton("green"));
    controls.add(blueButton = new JButton("blue"));
    controls.add(customButton = new JButton("custom"));
    topPane.add(controls, "South");
    // creates an event listener and registers it
    // with the buttons
    addListeners();
    selectColor(Color.red);
}

protected void addListeners() {
    // REQUIRES: The four button fields are not null.
    // MODIFIES: the buttons
    // EFFECTS: Creates an action event listener and
    // registers it with all four buttons.
    ActionListener l = new ActionListener() {
        public void actionPerformed(ActionEvent e) {
            Object source = e.getSource();
            if (source == redButton)
                selectColor(Color.red);
            else if (source == greenButton)
                selectColor(Color.green);
            else if (source == blueButton)
                selectColor(Color.blue);
            else selectColor();
        }
    };
    redButton.addActionListener(l);
    greenButton.addActionListener(l);
    blueButton.addActionListener(l);
    customButton.addActionListener(l);
}

protected void selectColor(Color color) {
    // REQUIRES: canvas is not null.
    // MODIFIES: color and canvas
    // EFFECTS: Sets the canvas background to color
    // and repaints it.
    this.color = color;
    canvas.setBackground(color);
    repaint();
}

protected void selectColor() {
```

```

// REQUIRES: canvas is not null.
// MODIFIES: color and canvas
// EFFECTS: Shows a color chooser dialog; if the
//   user chooses a color from the dialog then
//   selects the color; else does nothing.
Color newColor =
    JColorChooser.showDialog(null, "Choose Color", color);
if (newColor != null)
    selectColor(newColor);
}
}

```

练习

7.9 `java.awt.event.ActionEvent`类定义了下面的方法获得与动作事件相关的特定字符串：

```
public String getActionCommand();
```

`getActionCommand`返回的字符串称为事件的命令字串 (command string)。如果事件源为按钮，则命令字串是按钮的标签 (Label)。例如按red按钮，则返回字符串“red”。试重写`Color.addListener`方法的实现，以便匿名`ActionListener`通过事件的命令字串来区分按钮。

7.5.2 记录颜色

来看第二个程序`ColorRecord`，它扩展了`ColorPlay`的功能。除了四个颜色选择按钮之外，`ColorRecord`增加了一个颜色组合框，其中选项为一组代表不同颜色的图标，用户可以点选不同的图标来选择颜色。初始状态组合框只有三种颜色红绿蓝，随着程序不断运行，用户从弹出的颜色选择框中选择的颜色将不断加入到组合框颜色片中。图7-11为用户选择四种颜色后的情况。只有用户进行选择时组合框下拉列表才打开。

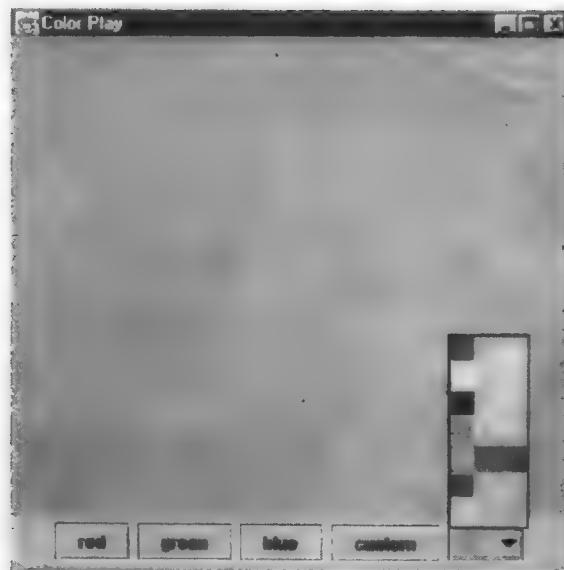


图7-11 Color Record程序

ColorRecord的实现中，方法selectColor是重载的。它有两个定义，第一个是按参数选择颜色，第二个是从弹出的颜色选择框中选择：

```
protected void selectColor(Color color);
protected void selectColor();
```

它们的功能和ColorPlay中的selectColor相同。ColorRecord又定义了三个管理颜色的方法，它们是：indexToColor，nbrColors，addColor。indexToColor由组合框中位置下标取得对应颜色值，下面的方法正是关心这一点的：

```
protected Color indexToColor(int colorIndx)
// REQUIRES: 0 <= colorIndx < nbrColors()
// EFFECTS: Returns the color at index colorIndx.
```

nbrColors方法返回已有颜色的数目：

```
protected int nbrColors()
// EFFECTS: Returns number of colors currently stored.
```

addColor方法增加新的颜色到组合框中：

```
protected void addColor(Color color)
// REQUIRES: color is not null.
// MODIFIES: vector colors, combo box colorsComboBox.
// EFFECTS: Adds color to the end of vector colors,
// and creates a chip for color and adds
// it to the end of colorsComboBox.
```

为实现上面三个方法，ColorRecord定义了一个域colors，它的值（Vector）用来按序保存组合框中的Color对象。下面程序是相关定义：

```
// fields of ColorRecord class
// combo box of color chips
protected JComboBox colorsComboBox;
// colors[i] stores the color of the chip at
// index i of colorsComboBox
protected Vector colors;

// methods of ColorRecord class
protected Color indexToColor(int colorIndx) {
    return (Color)colors.get(colorIndx);
}

protected int nbrColors() {
    return colors.size();
}

protected void addColor(Color color) {
    Icon chip = makeChip(CHIP_SIZE, CHIP_SIZE, color);
    colorsComboBox.addItem(chip);
    colors.add(color);
}
```

方法makeChip建立并返回一个指定大小和颜色的颜色片，颜色片是用ImageIcon对象表示的，为一张小图片。ImageIcon类实现了Icon接口，它定义了画固定宽和高的小图形的行为。下面是方法定义：

```
// method of ColorRecord class
protected Icon makeChip(int w, int h, Color color) {
```

```

// REQUIRES: w,h > 0, and color is not null.
// EFFECTS: Returns a new chip of width w, height h,
// and filled with color.
BufferedImage im =
    new BufferedImage(w,h, BufferedImage.TYPE_INT_RGB);
Graphics2D g2 = im.createGraphics();
g2.setPaint(color);
g2.fill(new Rectangle2D.Float(0, 0, w, h));
return new ImageIcon(im);
}

```

makeChip方法先创建指定大小BufferedImage对象，然后获得它的绘图环境g2，利用g2把BufferedImage填成指定颜色color，最后使用BufferedImage创建并返回ImageIcon对象。

下面是ColorRecord类的定义：

```

public class ColorRecord extends ApplicationFrame {

    protected JPanel canvas, controls;
    protected JButton redButton, greenButton,
                    blueButton, customButton;
    protected JComboBox colorsComboBox;
    protected Color color;
    protected Vector colors = new Vector();
    protected static final int CHIP_SIZE = 16;

    public static void main(String[] args) {
        JFrame frame = new ColorRecord("Color Record");
        frame.show();
    }

    public ColorRecord(String title) {
        super(title);
        Container topPane = getContentPane();
        topPane.setLayout(new BorderLayout());
        canvas = new JPanel();
        topPane.add(canvas, "Center");
        controls = new JPanel();
        controls.add(redButton = new JButton("red"));
        controls.add(greenButton = new JButton("green"));
        controls.add(blueButton = new JButton("blue"));
        controls.add(customButton = new JButton("custom"));
        colorsComboBox = new JComboBox();
        controls.add(colorsComboBox);
        topPane.add(controls, "South");
        addColor(Color.red);
        addColor(Color.green);
        addColor(Color.blue);
        selectColor(Color.red);
        addListeners();
    }

    protected void addListeners() {
        redButton.addActionListener(
            new ColorButtonListener(0));
        greenButton.addActionListener(
            new ColorButtonListener(1));
        blueButton.addActionListener(
            new ColorButtonListener(2));
    }
}

```

```

customButton.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        selectColor();
    }
});

colorsComboBox.addActionListener(new ActionListener(){
    public void actionPerformed(ActionEvent e) {
        int colorIndx = colorsComboBox.getSelectedIndex();
        selectColor(indexToColor(colorIndx));
    }
});
}

protected void selectColor(Color color) {
    this.color = color;
    canvas.setBackground(color);
    repaint();
}

protected void selectColor() {
    Color newColor =
        JColorChooser.showDialog(null, "Choose Color", color);
    if (newColor != null) {
        addColor(newColor);
        colorsComboBox.setSelectedIndex(nbrColors()-1);
    }
}

protected class ColorButtonListener
    implements ActionListener {
    int colorIndx;

    public ColorButtonListener(int colorIndx) {
        this.colorIndx = colorIndx;
    }

    public void actionPerformed(ActionEvent e) {
        colorsComboBox.setSelectedIndex(colorIndx);
    }
}

// the following methods were already defined
// in the text
protected void addColor(Color color) {...}
protected Color indexToColor(int colorIndx) {...}
protected int nbrColors() {...}
protected Icon makeChip(int w, int h, Color color) {...}
}

```

每个按钮都定义了自己的事件监听器，其中red、green和blue的事件监听器是同一个内部类ColorButtonListener的实例。ColorButtonListener的构造器的参数是颜色索引值。actionPerformed方法中设置颜色的索引值，同样就等于选择它的颜色和颜色片。

练习

7.10 假设内部类ColorButtonListener中的actionPerformed方法按下面方式实现：

```
public void actionPerformed(ActionEvent e) {
```

```

    selectColor(indexToColor(colorIdx));
}

```

那么程序ColorRecord的行为如何变化，试试看。

- 7.11 修改ColorPlay的实现，让每个按钮有自己的事件监听器，要求red、green、blue按钮的事件监听器由同一个类的实例观察，它的构造器带有color类型的参数。当监听器的actionPerformed被调用时，它选择它的颜色。
- 7.12 设计基于GUI程序SweepFigures，界面分成两个面板：南部为一个控制面板，中部为图形画布区。画布上用户绘制的图形和练习7.8中相似，不同之处在于图形可以是矩形或椭圆。控制面板上有一个按钮和一个组合框。clear按钮，用来清除画布上的所有图形；组合框中有两个选项：rectangle和ellipse，用户选择rectangle后，产生的图形为矩形，选择ellipse后，产生的图形为椭圆。
- 7.13 修改练习7.12中设计的程序SweepFigures，当用户在某些图形中按下鼠标按钮并拖动鼠标时，图形跟着鼠标焦点移动，直到鼠标释放，亦即用鼠标拖移图形（如果鼠标按下时有多个图形，则只移动其中一个图形）。其他功能同7.12程序。

7.6 点集三角形剖分程序：Triangulate

本节主要讨论一个由点集划分形成三角形剖分的基于GUI的程序例子（程序界面如7-12所示）。用户与程序交互主要有两种方式：（1）用户在程序画布上使用鼠标修改点集，这和7.2.5节中的EditPointSet程序相同：用户点击鼠标增加一个新点，拖动鼠标移动点，选中点按Ctrl删除点；（2）用户使用框架下部的控制面板上的按钮。

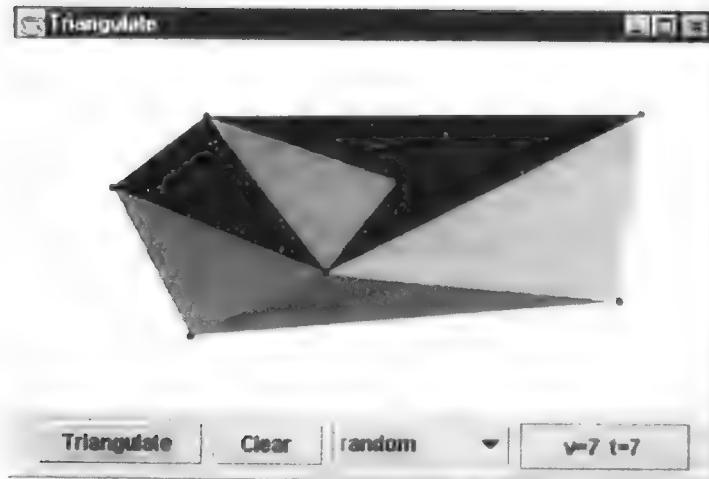


图7-12 Triangulate程序

控制面板上有两个按钮、一个标签和一个组合框，它们的作用分别为：用户点击Triangulate按钮，程序显示当前点集的三角形剖分；Clear按钮删除当前三角形剖分，但点集不受影响；最右边的一个为标签，v值为当前点数，t值为当前三角形剖分中的三角形数；剩下的是一个组合框（现在显示random选项），其中的选项决定由点集形成三角形时点的输入顺序（回忆练习6.18就可明白，点的输入顺序决定输出三角形的结果）。共有三

种选择：random选项随机排序输入点，user-defined选项按用户创建点的顺序排序输入点，sorted按点从左到右排序输入点。

如图 7-13 的程序类图结构所示, 程序设计中有两处需要强调说明。第一, `Triangulate` 对象框架包括两个面板: `TriangulateCanvas` 画布面板用于创建点和显示三角形剖分, `TriangulateControlPanel` 控制面板用于容纳控制按钮。第二, 7.2.5 节中 `EditPointSet` 作为本程序的一部分。实际上, 图 7-4 被嵌入图 7-13 中, 但有一点不同的是, 由于 `TriangulateManager` 增加了新的行为, 所以 `TriangulateCanvas` 可以和它直接交互, 而不需通过 `FigureManager` 接口。

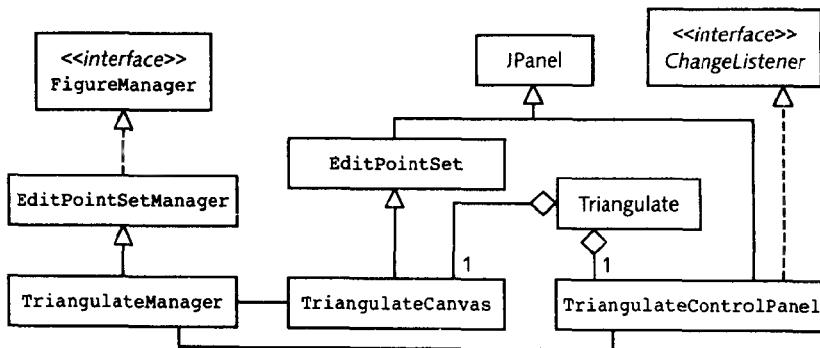


图7-13 Triangulate程序的结构

下面先从Triangulate类开始讨论我们的程序的实现。这个类表示一个框架，含有用边界布局管理器划分的两个面板，画布面板在中部区域，并占据框架的大部分；控制面板在南部区域，只占有足够放下它的组件的空间。下面为类定义：

```
public class Triangulate extends ApplicationFrame {  
  
    public static int D_WIDTH = 500, D_HEIGHT = 450;  
  
    public static void main(String[] args) {  
        JFrame frame =  
            new Triangulate("Triangulate", D_WIDTH, D_HEIGHT);  
        frame.show();  
    }  
  
    public Triangulate(String title,int width,int height) {  
        super(title, width, height);  
        Container topPane = getContentPane();  
        topPane.setLayout(new BorderLayout());  
        TriangulateCanvas canvas = new TriangulateCanvas();  
        TriangulateControlPanel controlPanel =  
            new TriangulateControlPanel(canvas.getManager());  
        topPane.add(canvas, "Center");  
        topPane.add(controlPanel, "South");  
    }  
}
```

TriangulateCanvas类负责点的修改和三角形剖分的图形绘制。其中点的修改由它的父类EditPointSet负责处理；而三角形剖分的图形绘制是由图形管理器处理。下面是它的类定义：

```

public class TriangulateCanvas extends EditPointSet {
    public void makeFigureManager(JPanel canvas) {
        manager = new TriangulateManager(canvas);
    }

    public TriangulateManager getManager() {
        return (TriangulateManager)manager;
    }
}

```

`makeFigureManager`方法是父类`EditPointSet`中定义的一个工厂方法。通过覆盖这个方法，`TriangulateCanvas`安装了自己的图形管理器。

下面讨论控制面板。图7-13中表明控制面板类继承实现了`ChangeListener`接口，`ChangeListener`定义如下：

```

public interface javax.swing.event.ChangeListener {
    // invoked when this listener's subject
    // undergoes a change in state
    public void stateChanged(ChangeEvent e);
}

```

一个事件变化监听器（change listener）注册到具有变化事件的事件源中，当事件源有状态的变化时，它会调用监听器的`stateChanged`方法通知所有已注册的监听器。这和本章前面所使用的事件模型是一致的。

像变化事件的监听器一样，控制面板注册到图形管理器。当面板中的三角形剖分发生变化时，图形管理器产生一个变化事件，相应的控制面板的`stateChanged`方法被调用。与此相似，当图形管理器创建一个新点或删除一个点时，它产生一个变化事件，同样`stateChanged`方法被调用。`stateChanged`方法同时负责刷新控制面板中显示当前点数和三角形数的标签值。

图7-14中所描述的顺序图是用户按`Triangulate`按钮后所发生的一系列事件（这里假设使用当前点集可以画出三角形剖分，即正常情况）：`Triangulate`按钮的事件监听器调用`addTriangulation`方法建立三角形剖分；如果调用正常返回，事件监听器刷新图形管理器；然后图形管理器刷新画布，并发`fireStateChanged`消息给自己，以提醒事件监听器自己状态的变化；相应地图形管理器调用控制面板的`stateChanged`方法。

如果用户按`Triangulate`按钮后不能正常形成三角形剖分（如当前点少于三个），`addTriangulation`会抛出异常，`Triangulate`按钮的事件监听器捕捉到这个信息，并刷新控制面板中标签信息，反映这个异常情况。图7-14图中没有画出异常部分。

控制面板类的实现定义几个域，一个保存到图形管理器的引用，其他四个域保存四个组件的每一个。下面为类定义：

```

public class TriangulateControlPanel extends JPanel
    implements ChangeListener {

    protected TriangulateManager manager; // figure manager
    protected JButton tButton,           // triangulate button
                  cButton;           // color button
    protected JLabel msgLabel;          // message label
    protected JComboBox pointOrderBox; // point ordering

    public TriangulateControlPanel(TriangulateManager mngr) {

```

```

    // register this panel as a listener of
    // the manager's change events
    this.manager = mngr;
    manager.addChangeListener(this);
    // create buttons, register their listeners, and
    // add them to panel
    add(tButton = new JButton("Triangulate"));
    tButton.addActionListener(new TriangulateListener());
    add(cButton = new JButton("Clear"));
    cButton.addActionListener(new ClearListener());
    // create point-order combo box and add it to
    // this panel
    pointOrderBox = new JComboBox();
    pointOrderBox.addItem("user-defined");
    pointOrderBox.addItem("random");
    pointOrderBox.addItem("sorted");
    add(pointOrderBox);
    // create message label and add it to this panel
    msgLabel = new JLabel();
    Border border =
        BorderFactory.createLineBorder(Color.blue);
    msgLabel.setBorder(border);
    msgLabel.setHorizontalAlignment(SwingConstants.CENTER);
    msgLabel.setPreferredSize(tButton.getPreferredSize());
    updateMessage();
    add(msgLabel);
}

//
// implement ChangeListener interface
//
public void stateChanged(ChangeEvent e) {
    updateMessage();
}

protected void updateMessage() {
    // MODIFIES: msgLabel
    // EFFECTS: Sets label's text to number of points
    // and triangles.
    String res = "v=" + manager.nbrVertices();
    int nbrTriangles = manager.nbrTriangles();
    if (nbrTriangles > 0)
        res += " t=" + nbrTriangles;
    msgLabel.setForeground(Color.black);
    msgLabel.setText(res);
}

protected void updateMessage(String msg) {
    // MODIFIES: msgLabel
    // EFFECTS: Sets label's text to msg.
    msgLabel.setForeground(Color.red);
    msgLabel.setText(msg);
}

//
// action listener classes for the Triangulate and
// Clear buttons
//
class TriangulateListener implements ActionListener {
    public void actionPerformed(ActionEvent evt) {
        try {
            String pointsOrdering =
                (String)pointOrderBox.getSelectedItem();
            manager.addTriangulation(pointsOrdering);
        }
    }
}

```

```

        manager.updateManager();
    } catch (IllegalArgumentException e) {
        updateMessage("Too few points");
    } catch (ColinearPointsException e) {
        updateMessage("Colinear points");
    }
}

class ClearListener implements ActionListener {
    public void actionPerformed(ActionEvent evt) {
        manager.removeTriangulation();
        manager.updateManager();
    }
}
}
}

```

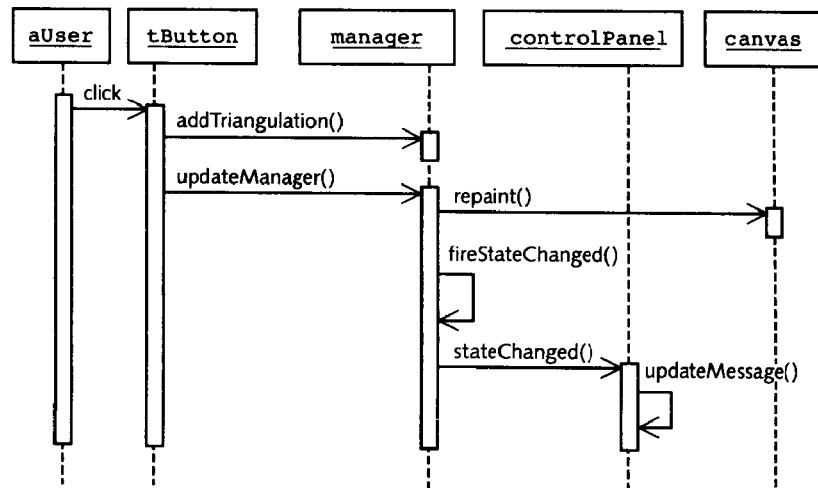


图7-14 创建三角形剖分的顺序图

`TriangulateManager`类管理两类图形：用户创建的点集和当前三角形剖分中的三角形。点集管理功能是由它的父类`EditPointSetManager`提供的，而对三角形的管理功能是由`TriangulateManager`自己提供的。三角形都保存在`GroupNode`类型的`Triangulation`域中，三角形是组节点的子节点。当响应`getFigures`消息时，如果没有三角形存在，则返回包含点集的对象；否则，创建一个带有两个子节点`GroupNode`对象，一个子节点是继承的点集对象，另一个是三角形对象。

`TriangulateManager`类中同时定义了`changeListener`域，它指向惟一注册的事件监听器。当它收到`updateManager`消息时，会调用`fireStateChanged`方法，通过它调用事件监听器的`stateChanged`方法。这个过程在图7-14中有表示。下面是类定义：

```

public class TriangulateManager
    extends EditPointSetManager {

    protected GroupNode triangulation;
    protected ChangeListener changeListener;

    public TriangulateManager(JPanel canvas) {
        super(canvas);
    }
}

```

```

//  

// override methods of FigureManager interface  

//  

public void updateManager() {  

    super.updateManager();  

    fireStateChanged();  

}  

public Node getFigures() {  

    if (triangulation == null)  

        return node;  

    else {  

        GroupNode topNode = new GroupNode();  

        topNode.addChild(triangulation);  

        topNode.addChild(node);  

        return topNode;  

    }  

}  

//  

// adding a change listener and implementing  

// ChangeListener interface  

//  

public void addChangeListener(ChangeListener listener){  

    changeListener = listener;  

}  

protected void fireStateChanged() {  

    if (changeListener != null)  

        changeListener.stateChanged(new ChangeEvent(this));  

}  

//  

// methods used by TriangulateControlPanel class  

//  

public void addTriangulation(String pointOrdering)  

    throws IllegalArgumentException,  

        ColinearPointsException {  

    // REQUIRES: node, triangulation, pointOrdering are  

    // not null.  

    // MODIFIES: this  

    // EFFECTS: If points contains no more than two  

    // points throws IllegalArgumentException; else  

    // if the first three points are colinear throws  

    // ColinearPointsException; else triangulates  

    // by ordering the input points according to  

    // pointOrdering and adds the triangles to  

    // the node triangulation.  

    PointGeometry[] points = extractPoints();  

    arrangePoints(points, pointOrdering);  

    Vector triangles =  

        DynamicPolygons.triangulation(points);  

    triangulation = new GroupNode();  

    for (int i = 0; i < triangles.size(); i++) {  

        Geometry g = (Geometry)triangles.get(i);  

        Painter painter = new FillPainter(rnd.nextInt());  

        triangulation.addChild(new Figure(g, painter));  

    }  

    updateManager();  

}  

public void removeTriangulation() {  

}

```

```

// MODIFIES: this
// EFFECTS: Removes the triangulation and
//   updates the manager.
triangulation = null;
updateManager();
}

public int nbrVertices() {
    // EFFECTS: Returns the number of points
    //   in the point set.
    return node.nbrChildren();
}

public int nbrTriangles() {
    // EFFECTS: If triangulation exists returns the
    //   number of triangles; else returns zero.
    if (triangulation != null)
        return triangulation.nbrChildren();
    else return 0;
}

//
// protected interface
//
protected PointGeometry[] extractPoints() {
    // REQUIRES: node is not null.
    // EFFECTS: Returns an array of the points
    //   in the point set.
    PointGeometry[] points =
        new PointGeometry[node.nbrChildren()];
    for (int i = 0; i < node.nbrChildren(); i++)
        points[i] = (PointGeometry) get(i);
    return points;
}

protected void arrangePoints(PointGeometry[] points,
                            String ordering) {
    // REQUIRES: points, ordering is not null.
    // MODIFIES: points
    // EFFECTS: Orders points according to the protocol
    //   ordering; does nothing if ordering is neither
    //   "sorted" nor "random".
    if (ordering.equals("sorted"))
        Arrays.sort(points);
    else if (ordering.equals("random"))
        Collections.shuffle(Arrays.asList(points));
}
}

```

练习

7.14 `Triangulate`程序输出的点的颜色是随机的，请修改程序使点的输出颜色为白色。
 [提示：覆盖`TriangulateManager`类的`add`方法。]

7.15 前面`Triangulate`程序中，点的输出在三角形剖分的上面。请修改程序，使三角形剖分的输出在点的上面。

7.16 为何没有必要让`Clear`按钮的事件监听器调用`updateMessage`方法？消息标签是如何更新的？

7.7 画图程序：DrawPad

在这一节，将设计一个用于绘画和修改图形的基于GUI的程序DrawPad。DrawPad的框架包括两个面板：包含几个按钮的控制面板和可以在上面画图的画布面板。控制面板上有三类按钮：形状按钮（shape button）、指针工具按钮（pointer button）和颜色按钮（color button）。如图7-15所示，形状按钮包括Polygon、Rectangle和Ellipse，用它们可以产生和修改不同形状的图形；指针工具按钮，用户按了它之后，再点选某个图形，就选中这个图形并把它放到前景中（使图形有效，图形的周围出现高亮度的选中框，见图7-15的椭圆周围的框），然后可以拖动图形来移动它，或按Ctrl删除图形。按颜色按钮后，弹出一个颜色选择对话框来选择新颜色，新颜色被用于选中图形。关于画图程序的功能会随着讨论不断说明。

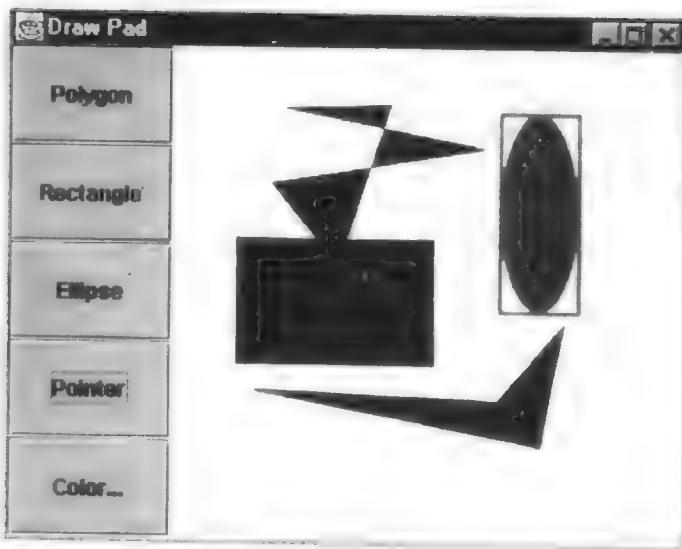


图7-15 DrawPad程序

DrawPad程序的设计包括三个类图。第一个类图（图7-16）主要描述程序的GUI组件和图形管理器；第二个类图（图7-18）主要描述各种事件监听器；第三个类图（图7-20）主要描述各种不同的高亮度选中图形的策略。三个类图之间是用Tool接口联系起来的，每个画布事件监听器都实现Tool接口，用来处理画布上的鼠标事件。下面分三个部分分别讨论：7.7.1节主要是介绍DrawPad的组件和图形管理器；7.7.2节讨论它的事件监听器（Tool接口和它的子型）；7.7.3节介绍高亮度图形的策略。

7.7.1 DrawPad的组件和图形管理器

DrawPad是一个内容格被分成两个面板的框架：控制面板类ControlPanel，它的按钮被用于管理图形和选择画图工具；画布面板类DrawCanvas，用户在其中画图形和编辑图形，参见图7-16。当用户按其中一个控制面板的按钮时，它会注册一个相应的鼠标事件监听器，用来响应用户的操作。例如用户按Rectangle后，它会注册一个把用户的操作翻译成对矩形的操纵的事件监听器。所有的事件监听器都要实现Tool接口。程序使用

一个DrawPadManager对象来管理图形。DrawPadManager类实现了接口FigureManager。为了重画，画布从图形管理器获得当前图形集。

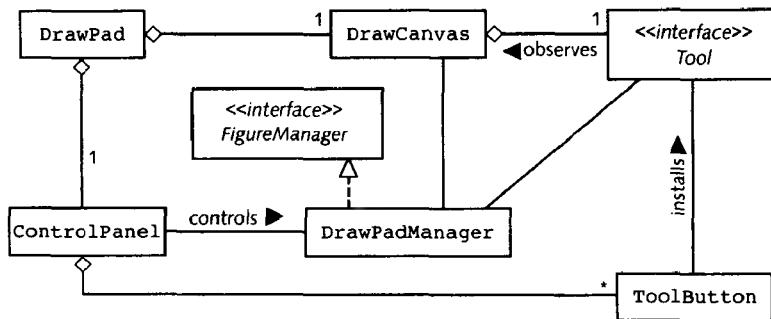


图7-16 DrawPad程序中的组件

在这一节中，我们将实现图7-16中的设计。先看类DrawPad，它定义了一个静态的main方法，DrawPad是一个包含控制面板和画布的框架：

```

public class DrawPad extends ApplicationFrame {

    public static final int DEFAULT_WIDTH = 500,
                        DEFAULT_HEIGHT = 450;

    protected DrawCanvas canvas;

    public static void main(String[] args) {
        JFrame frame =
            new DrawPad("DrawPad", DEFAULT_WIDTH, DEFAULT_HEIGHT);
        frame.show();
    }

    public DrawPad(String title, int width, int height) {
        super(title, width, height);
        Container topPane = getContentPane();
        topPane.setLayout(new BorderLayout());
        topPane.add(canvas = new DrawCanvas(), "Center");
        topPane.add(new ControlPanel(canvas), "West");
    }
}
  
```

利用边界布局管理器，包括五个按钮的控制面板竖放在框架左部。每个形状按钮和指针按钮都由一个ToolButton对象表示。它们在创建时就把相应的事件监听器注册，这也是为什么下面的ControlPanel构造器中没有单独对按钮进行事件监听器注册的原因。而对颜色按钮就不同，由于它不是ToolButton对象，所以要调用AddActionListener方法注册事件监听器。下面是ControlPanel类定义：

```

public class ControlPanel extends JPanel {

    protected DrawPadManager manager;
    protected JButton colorButton;
    public ControlPanel(DrawCanvas canvas) {
        this.manager = canvas.getManager();
        ToolButton polyButton;
        setLayout(new GridLayout(5, 1));
        add(polyButton = new ToolButton("Polygon",
  
```

```

        new PolygonTool(canvas)));
add(new ToolButton("Rectangle",
                   new RectangleTool(canvas)));
add(new ToolButton("Ellipse",
                   new EllipseTool(canvas)));
add(new ToolButton("Pointer",
                   new PointerTool(canvas)));
// install the polygon tool initially
polyButton.doClick();
// color button
colorButton = new JButton("Color...");
colorButton.setBackground(manager.getColor());
add(colorButton);
colorButton.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        Color oldColor = manager.getColor();
        Color newColor =
JColorChooser.showDialog(null, "Choose Color",
                           oldColor);
        if (newColor != null) {
            manager.setColor(newColor);
            colorButton.setBackground(newColor);
        }
    }
});
```

在上面ControlPanel构造器中，颜色按钮colorButton的事件监听器是一个实现了ActionListener接口的匿名内部类的对象。当用户点击颜色按钮时，它调用JColorChooser.showDialog弹出颜色选择对话框，然后用户选择需要的颜色并关闭对话框，JColorChooser.showDialog返回所选的颜色（如果用户取消操作并关闭对话框，则返回null），接着监听器通知图形管理器新的颜色并设置颜色按钮的背景为新颜色（按钮背景颜色用于通知用户当前颜色）。

ToolButton是一个与工具相关的按钮（工具是处理画布上图形形状的事件监听器）；当用户点击工具按钮时，它的事件监听器会安装它的工具，用它来作为处理画布上图形的事件监听器。下面是**ToolButton**的实现：

```
public class ToolButton extends JButton {  
  
    protected Tool tool;  
  
    public ToolButton(String text, Tool tool) {  
        super(text);  
        this.tool = tool;  
        addActionListener(new ToolButtonListener());  
    }  
  
    class ToolButtonListener implements ActionListener {  
        public void actionPerformed(ActionEvent e) {  
            tool.install();  
        }  
    }  
}
```

图7-17的顺序图展示了当用户按工具按钮来选择新的工具时发生的一连串交互过程：theNewTool收到install消息，然后把install消息传给画布，并把自己作为参数传递。作为

回应，画布取消 (`finish`) 原来 `theOldTool` 作为鼠标事件监听器的资格，并释放它；然后把 `theNewTool` 注册为新的鼠标事件监听器，并进行初始化 (`init`)。这个过程在定义 `Tool` 接口和 `DrawCanvas` 类中会详细说明。

`DrawPad` 程序的画布面板由 `DrawCanvas` 类实现。如图 7-17 所示，`DrawCanvas` 类定义了 `install` 方法，在这个方法中取消原来鼠标事件监听器并注册新的事件监听器。同时 `DrawCanvas` 类还覆盖了用来输出图形当前集的 `paintComponent` 方法。`getManager` 方法用来获得到图形管理器的引用。下面是 `DrawCanvas` 类定义：

```
public class DrawCanvas extends JPanel {
    public static final Color DefaultColor = Color.blue;
    protected Tool currentTool;
    protected DrawPadManager manager;
    protected Figure currentFigure;

    public DrawCanvas() {
        setBackground(Color.black);
        currentTool = null;
        manager = new DrawPadManager(this);
        manager.setColor(DefaultColor);
    }

    public DrawPadManager getManager() {
        return manager;
    }

    public boolean install(Tool tool) {
        if (currentTool == tool)
            return false;
        else if (currentTool != null) {
            currentTool.finish();
            removeMouseListener(currentTool);
            removeMouseMotionListener(currentTool);
        }
        currentTool = tool;
        addMouseListener(currentTool);
        addMouseMotionListener(currentTool);
        currentTool.init();
        return true;
    }

    public void paintComponent(Graphics g) {
        super.paintComponent(g);
        Graphics2D g2 = (Graphics2D)g;
        g2.setRenderingHint(RenderingHints.KEY_ANTIALIASING,
                           RenderingHints.VALUE_ANTIALIAS_ON);
        manager.getFigures().paint(g2);
    }
}
```

下面来看 `DrawPadManager` 类。它的主要功能是对图形进行管理，它实现了 7.2.5 节中的 `FigureManager` 接口。对 `DrawPad` 程序来说，图形管理器的职责是什么呢？图形管理器主要负责管理一个图形集合，按它们输出的先后顺序排列。在这种分层排列 (Layer Ordering) 的结构中，每个图形都有自己的层，输出时图形从最底层到最上层依次进行，所以有可能前面图形挡住了后面的图形。如图 7-15 中的五个顶点多边形就在矩形的上面。

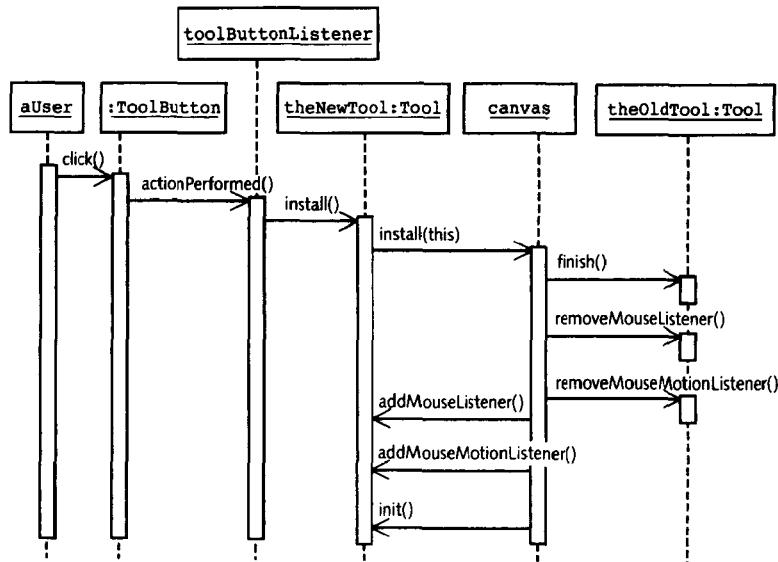


图 7-17 在画布上安装工具的顺序图

图形管理器的另一个功能是负责跟踪所选中的几何图形和它属于的图形。当响应 `getFigures` 消息时，管理器会返回所有的图形包括高亮度选中的图形。因为不同形状的图形的高亮度选中方法不同，所以策略不同，使用 `hilightStrategy` 域中保存的策略来建立选中图形不同的策略。

`DrawPadManager` 类定义了五个域：`canvas` 域保存画布对象引用；`node` 域按层的顺序保存其子节点是图形的组节点（也就是，发送给 `node` 一个 `paint` 消息自底向上绘制图形）；`currentFigure` 域保存到选中图形的引用；`currentColor` 域保存当前颜色；`hilightStrategy` 域保存高亮度图形的当前策略。下面是 `DrawPadManager` 类的定义，其中有几个方法的实现省略掉，留在练习中完成：

```

public class DrawPadManager implements FigureManager {

    protected DrawCanvas canvas;
    protected GroupNode node;
    protected Figure currentFigure;
    protected Color currentColor;
    protected HilightStrategy hilightStrategy;
    public DrawPadManager(DrawCanvas canvas) {
        this.canvas = canvas;
        node = new GroupNode();
    }

    public void select(Geometry g)
        throws IllegalArgumentException {
        // MODIFIES: this
        // EFFECTS: If g is not in this collection throws
        //          IllegalArgumentException; else if g is not null
        //          makes g the current geometry and moves its
        //          figure to the topmost layer; else no geometry
        //          is made current.
        if (g == null) {
            currentFigure = null;
        }
    }
}
  
```

```

        return;
    }
    int indx = findFigureIndex(g);
    if (indx < 0) throw new IllegalArgumentException();
    else {
        currentFigure = (Figure)node.child(indx);
        node.removeChild(indx);
        node.addChild(currentFigure);
    }
}

public Geometry selected() {
    // EFFECTS: Returns the current geometry if any;
    //   else returns null.
    if (currentFigure != null)
        return currentFigure.getGeometry();
    else return null;
}

public Node getFigures() {
    // EFFECTS: Returns a node whose children
    //   represent the set of figures,
    //   including any highlights.
    GroupNode n = new GroupNode();
    n.addChild(node);
    Node hilight =
        hilightStrategy.makeHilight(selected());
    if (hilight != null)
        n.addChild(hilight);
    return n;
}

public void updateManager() {
    // EFFECTS: Repaints the canvas.
    canvas.repaint();
}

public void add(Geometry g) {
    // MODIFIES: this
    // EFFECTS: Creates a new figure with geometry g
    //   and a painter based on the current color
    //   and adds figure at the top layer.
    Painter painter = makePainter(getColor());
    node.addChild(new Figure(g, fill));
}

public void remove(Geometry g) {
    // MODIFIES: this
    // EFFECTS: Removes from this collection
    //   a figure with geometry g;
    //   does nothing if no such figure exists.
    int indx = findFigureIndex(g);
    if (indx >= 0) {
        node.removeChild(indx);
        if (selected() == g)
            select(null);
    }
}

public Geometry get(int i)
    throws IndexOutOfBoundsException {
    // EFFECTS: Returns the figure at index i.
}

```

```

Figure fig = (Figure)node.child(i);
return fig.getGeometry();
}

public int size() {
    // EFFECTS: Returns the number of figures
    // in this collection.
    return node.nbrChildren();
}

public Geometry find(int x, int y) {
    // EFFECTS: Returns the geometry that contains the
    // point(x,y) whose figure lies in the highest
    // layer; returns null if no such geometry exists.
    ...
}

public Color getColor() {
    // EFFECTS: Returns the current color.
    ...
}

public void setColor(Color newColor) {
    // MODIFIES: this
    // EFFECTS: Sets the current color to newColor,
    // and if there exists a selected figure,
    // changes its color to newColor.
    ...
}

public void setHilightStrategy(HilightStrategy s) {
    // REQUIRES: s is not null.
    // MODIFIES: this
    // EFFECTS: Sets the hilight strategy to s.
    this.hilightStrategy = s;
}

protected int findFigureIndex(Geometry g) {
    // EFFECTS: Returns the index of the figure
    // whose geometry is g if
    // such a figure exists; else returns -1.
    ...
}

protected Painter makePainter(Color color) {
    // EFFECTS: Returns a new painter based on color.
    Painter draw = new DrawDynamicPolygonPainter(color);
    Painter fill = new FillPainter(color);
    return new MultiPainter(fill, draw);
}
}

```

可以观测到，上面DrawPadManager类的方法的后置条件比FigureManager接口指定的后置条件更强。例如，FigureManager.find记录由输入点(x, y)“命中”的几何图形，而DrawPadManager.find记录了包含(x, y)的最上面的几何图形。还要注意，除了由FigureManager接口指定的方法外DrawPadManager还实现了三个公有方法getColor、setColor和setHilightStrategy。

getFigure方法的定义很有趣。它不是仅仅返回由node域引用的组节点，而是返回一个新的组节点，其中第一个子节点为node，而第二个子节点为高亮度选中图形的节点。

node的状态不会被该操作改变。

练习

7.17 完成DrawPadManager类的实现。

7.7.2 DrawPad的事件监听器

画布上用户的动作——鼠标点击或鼠标拖动——可以被当时画布上的任何一个工具解释。例如，如果按绘图工具（pointer）后，鼠标点击用于选中图形；如果多边形工具被激活，鼠标点击会创建顶点或删除顶点。本节中将详细讨论图7-18中的Tool接口和由它产生的子型。

```
public interface Tool extends MouseListener,
                           MouseMotionListener {
    public abstract void install();
    // EFFECTS: Installs this tool in the canvas.

    public abstract void init();
    // MODIFIES: this
    // EFFECTS: Any (initializes this tool).

    public abstract void finish();
    // MODIFIES: this
    // EFFECTS: Any (cleans up this tool).

    public abstract HilightStrategy makeHilightStrategy();
    // MODIFIES: this
    // EFFECTS: Returns a highlighting strategy
    // appropriate to the type
    // of geometries handled by this tool.
}
```

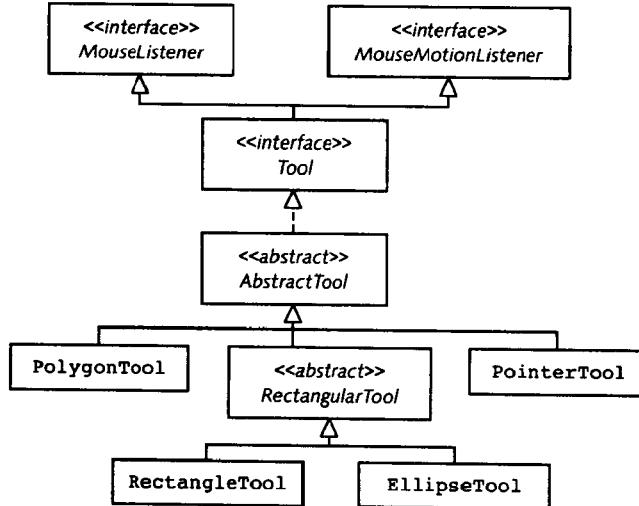


图7-18 Tool接口和它的子型

图7-17所示的顺序图表明当一个新工具被安装时，init和finish方法所起的作用。注意这些方法的后置条件没有规定工具以何种方式初始化或终止自身。

抽象类AbstractTool实现了接口Tool，在该类中实现了工具的大部分行为。它提

供了两个域：一个用来保存它监听的画布的引用，另一个是保存它使用的图形管理器的引用。**AbstractTool**类还提供了**install**方法的默认行为：安装自己的对象到画布中，如果安装失败，设置当前几何图形为**null**并给该工具重新初始化自身的机会。**init**方法的默认行为是通知图形管理器它所使用的高亮度选中策略。其他的方法的实现为默认，没有做任何事情。下面为抽象类**Abstract Tool**的定义：

```
public abstract class AbstractTool implements Tool {

    protected DrawCanvas canvas;
    protected DrawPadManager manager;

    protected AbstractTool(DrawCanvas canvas) {
        this.canvas = canvas;
        this.manager = canvas.getManager();
    }

    public void install() {
        if (!canvas.install(this)) {
            finish();
            manager.select(null);
            init();
        }
    }

    public void init() {
        manager.setHilightStrategy(makeHilightStrategy());
    }

    public void finish() { }
    public void mouseEntered(MouseEvent e) { }
    public void mouseExited(MouseEvent e) { }
    public void mouseClicked(MouseEvent e) { }
    public void mouseMoved(MouseEvent e) { }
}
```

AbstractTool的子类必须实现父类中没有实现的其他鼠标监听器方法(**mousePressed**、**mouseReleased**和**mouseDragged**)和**Tool**接口声明的工厂方法**makeHilightStrategy**。注意，**AbstractTool**的子类通过保护型域**canvas**和**manager**可以访问画布和图形管理器。

下面先从子类**PointerTool**开始**AbstractTool**的子型的定义。**PointerTool**用于选择、移动和删除图形。下面是这些行为的面向用户描述：

- 当用户点击指针工具时，如果当前有被选中的图形，则此图形被释放，并且光标变为箭头图标（相反，当形状定义工具被激活时，光标变为十字图标）。
- 如果用户点击任何图形，则它被选中，在它周围加上一个高亮度显示框；如果用户点击图形区外的其他背景地方，当前选中图形被释放。
- 当用户拖动一个图形后，图形随着光标移动；当释放鼠标时，图形被定位在最后的位置。但如果当他按着**Ctrl**键时释放鼠标时，图形会被删除。

下面是类**PointerTool**的定义：

```
public class PointerTool extends AbstractTool {

    protected Geometry currentGeometry;
    protected PointGeometry lastPoint;

    public PointerTool(DrawCanvas canvas) {
```

```

        super(canvas);
    }

    public void init() {
        super.init();
        manager.select(null);
        canvas.setCursor(Cursor.getDefaultCursor());
        manager.updateManager();
    }

    public HilightStrategy makeHilightStrategy() {
        return new OutlineStrategy();
    }

    public void mousePressed(MouseEvent e) {
        currentGeometry = manager.find(e.getX(), e.getY());
        manager.select(currentGeometry);
        if (currentGeometry != null)
            lastPoint = new PointGeometry(e.getX(), e.getY());
        manager.updateManager();
    }

    public void mouseDragged(MouseEvent e) {
        if (currentGeometry != null) {
            int x = e.getX();
            int y = e.getY();
            int dx = x - lastPoint.getX();
            int dy = y - lastPoint.getY();
            currentGeometry.translate(dx, dy);
            lastPoint.setX(x);
            lastPoint.setY(y);
            manager.updateManager();
        }
    }

    public void mouseReleased(MouseEvent e) {
        if ((currentGeometry != null) && e.isControlDown()) {
            manager.remove(currentGeometry);
            currentGeometry = null;
        }
        lastPoint = null;
        manager.updateManager();
    }
}

```

类PointerTool的域currentGeometry用来保存当前选中的几何图形的引用。域lastPoint用来在鼠标拖动时移动当前几何图形，该域记录最近鼠标点位置。当鼠标被拖动时，mouseDragged方法计算移动的量，算法是鼠标当前点减去鼠标最近点，得到移动的dx和dy距离。每个鼠标事件的处理最后都调用updateManager方法，用来更新图形显示。makeHilightStratety方法的实现返回一个高亮度显示策略。它和其他的策略将在7.7.3节中详述。

下面我们考虑画矩形形状（矩形和椭圆形）的工具。一般情况下，画形状工具用于画新图形或编辑当前图形。在画矩形形状的工具的情况下，用户管理图形定界框的拐角。下面是对所有矩形工具通用的行为：

- 当用户选中画形状工具时，如果有选中图形并且选中图形类型和该工具类型一致，则选中图形不变；反之，则选中图形被释放，并且光标变为十字图标。
- 当一个图形被选中后，用户可拖着它的任一个角到新的位置。在这期间，它的对角

则保持在原地不动。这就是此类图形的重定型规则。

- 被选中图形的周围有一个高亮度矩形定界框，这个框对椭圆重定型时定位它的“角”很有用。
- 如果没有选中图形，用户可以画出一个新图形，先定一个点（角），然后拖鼠标到另一个位置，点一下确定对角。
- 如果产生矩形几何图形的宽高都为0，该几何图形被删掉。

类RectangularTool的定义相比较有些复杂，但是从下面两个因素考虑就会减轻这种感觉。首先，它的定义复杂，它的子型的定义就会简单些；其次，在这个设计中，使用了两种设计模式：模板设计模式和工厂方法设计模式（见图7-19）。RectangularTool类的init方法是一个模板方法。这个方法的一个功能是判断它是否可以修改选中的图形，调用抽象方法canEdit来实现。抽象方法canEdit是由子类实现的。这里模板方法(init)定义了一个算法，该算法的一个步骤(canEdit)由子类实现。

makeRectangularGeometry是一个工厂方法。它由mousePressed方法调用，用来创建图形对象。但由于mousePressed方法并不清楚要创建的几何图形的实际类型，所以这一决策留给makeRectangularGeometry方法的子类实现。因此，RectangularTool类提供了一个创建新对象的接口，让它的子类来决定要创建哪一类型的对象。从图7-19可以看出，我们使用了这两种模式。

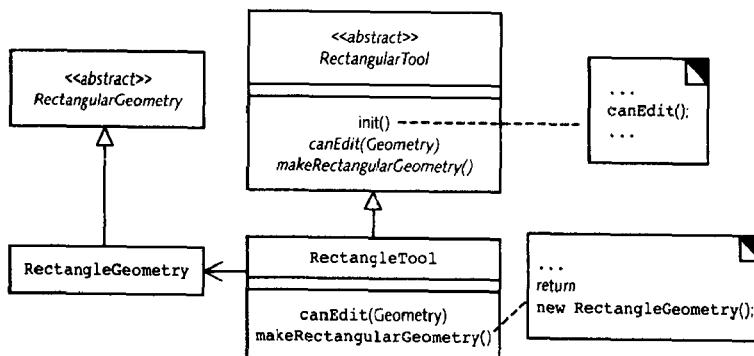


图7-19 矩形工具中应用的模板方法模式和工厂方法模式

下面类RectangularTool的实现定义有两个域：geometry域保存当前工具正在编辑的几何图形；anchor域用在创建图形和重定型已存在的图形中。在这两种情况下，anchor都用来保存对角的位置。下面是RectangularTool的类定义：

```

public abstract class RectangularTool
    extends AbstractTool {

    protected RectangularGeometry geometry;
    protected PointGeometry anchor;

    public RectangularTool(DrawCanvas canvas) {
        super(canvas);
    }

    public void init() {
        super.init();
        canvas.setCursor(Cursor.getPredefinedCursor(
  
```

```

        Cursor.CROSSHAIR_CURSOR));
Geometry geom = manager.selected();
if (!canEdit(geom)) {
    manager.select(null);
    geometry = null;
} else
    geometry = (RectangularGeometry)geom;
manager.updateManager();
}

public void finish() {
if ((geometry != null) &&
    ((geometry.getWidth() == 0) ||
     (geometry.getHeight() == 0))) {
    manager.select(null);
    manager.remove(geometry);
    geometry = null;
}
}

public HilightStrategy makeHilightStrategy() {
    return new BoundingBoxStrategy();
}

//  

// abstract methods  

//  

public abstract RectangularGeometry
    makeRectangularGeometry(PointGeometry p,int w,int h);
// REQUIRES: p is not null,
// and w and h are nonnegative.
// EFFECTS: Returns a new rectangular geometry with
// position p, width w, and height h.

public abstract boolean canEdit(Geometry geom);
// EFFECTS: Returns true if this tool is capable
// of editing geom; returns false if geom is
// null or otherwise cannot edit geom.

public void mousePressed(MouseEvent e) {
    PointGeometry firstPoint =
        new PointGeometry(e.getX(), e.getY());
    if (geometry == null) {
        anchor = firstPoint;
        geometry = makeRectangularGeometry(anchor, 0, 0);
        manager.add(geometry);
        manager.select(geometry);
    } else
        anchor = oppositeCorner(firstPoint);
    manager.updateManager();
}

public void mouseDragged(MouseEvent e) {
    if (anchor != null) {
        PointGeometry curPoint =
            new PointGeometry(e.getX(), e.getY());
        updateRectangularGeometry(anchor, curPoint);
        manager.updateManager();
    }
}

public void mouseReleased(MouseEvent e) {
    if (anchor == null) return;
    PointGeometry curPoint =
        new PointGeometry(e.getX(), e.getY());
    updateRectangularGeometry(anchor, curPoint);
}

```

```

protected void
updateRectangularGeometry(PointGeometry p1,
                           PointGeometry p2) {
    // REQUIRES: p1, p2, and geometry are not null.
    // MODIFIES: geometry
    // EFFECTS: Updates the dimensions of geometry such
    // that p1 and p2 are made opposite corners.
    int x = Math.min(p1.getX(), p2.getX());
    int y = Math.min(p1.getY(), p2.getY());
    int width = Math.abs(p2.getX() - p1.getX());
    int height = Math.abs(p2.getY() - p1.getY());
    geometry.setPosition(new PointGeometry(x, y));
    geometry.setWidth(width);
    geometry.setHeight(height);
}

protected PointGeometry
oppositeCorner(PointGeometry p) {
    // REQUIRES: p and geometry are not null.
    // EFFECTS: Returns null if p is not near (within 3
    // pixels of) one of geometry's corners; else
    // where p is near corner c, returns the corner
    // of geometry that is opposite c.
    PointGeometry res = new PointGeometry();
    int minX = geometry.getPosition().getX();
    int maxX = minX + geometry.getWidth();
    Range xRange = new Range(p.getX()-3, p.getX()+3);
    if (xRange.contains(minX)) res.setX(maxX);
    else if (xRange.contains(maxX)) res.setX(minX);
    else return null;
    int minY = geometry.getPosition().getY();
    int maxY = minY + geometry.getHeight();
    Range yRange = new Range(p.getY()-3, p.getY()+3);
    if (yRange.contains(minY)) res.setY(maxY);
    else if (yRange.contains(maxY)) res.setY(minY);
    else return null;
    return res;
}
}

```

看一下上面程序类RectangularTool两个抽象方法的调用。makeRectangularGeometry工厂方法是在没有选中的几何图形时（也就是geometry==null为真时），由mousePressed方法调用。这段代码是在用户按下鼠标创建新的图形（矩形或椭圆）时执行的。canEdit方法由init方法在矩形工具被安装到画布时调用。init方法使用canEdit来判断该工具是否能编辑选中图形，如果不能，则选中图形被释放并从头开始配置工具来构造新的图形。

RectangularTool的具体子类负责实现抽象makeRectangularGeometry和canEdit方法，下面是子类RectangleTool的定义：

```

public class RectangleTool extends RectangularTool {
    public RectangleTool(DrawCanvas canvas) {
        super(canvas);
    }

    public RectangularGeometry
    makeRectangularGeometry(PointGeometry p, int w, int h) {
        return new RectangleGeometry(p, w, h);
    }

    public boolean canEdit(Geometry g) {

```

```

        return (g instanceof RectangleGeometry);
    }
}

```

最后我们看创建和编辑多边形的 **PolygonTool** 类。这个类的行为和 7.2.4 节中的多边形图形修改监听器很相似，不同的是这个类不仅可以修改，还可以创建和删除多边形。下面为 **PolygonTool** 面向用户行为的描述：

- 当用户点击多边形工具时，如果有选中图形并且图形类型为多边形，则选中图形不变；反之，则选中图形被释放，并且光标变为十字图标。
- 当一个多边形被选中时，用户用鼠标点击背景任何位置，则多边形增加一个新顶点，新增点变成焦点所在的当前点；按 Ctrl 键时，当前点被删除，它的前一个点变成当前点；如果只有一个点时，则多边形被删除。鼠标选中一顶点拖动时，这个点移动到新的位置。
- 被选中多边形的轮廓被高亮度显示，当前点以明亮的色彩显示。
- 如果没有选中图形，用户用鼠标点击背景某一位置，在点击点处创建一个顶点的多边形。
- 如果所产生多边形的顶点少于三个时，则被删掉。

PolygonTool 类的实现和 7.2.4 节中的 **PolygonListener** 很相似。它们都定义两个域：**iter** 域保存选中多边形的多边形迭代器，用来遍历所选中多边形的顶点；布尔型的 **vertexBeingDragged** 域用来标识用户是否用鼠标拖着顶点移动。尽管如此，它们还是有下面三个区别：第一，作为一种工具 **PolygonTool** 扩展了类 **AbstractTool**；第二， **PolygonTool** 不像 **PolygonListener** 那样要自己刷新图形，而是使用图形管理器（它从它的父类 **AbstractTool** 继承了 **manager** 域）来完成；第三， **PolygonTool** 的保护型接口的定义要更详尽一些，因为它不仅要完成图形修改，还要负责图形创建和删除。有趣的是，它们的鼠标事件处理方法 **mousePressed**、**mouseReleased** 和 **mouseDragged** 基本相同，只是它们所调用的保护型方法的实现不同。下面是 **PolygonTool** 类的类定义：

```

public class PolygonTool extends AbstractTool {

    protected PolygonIterator iter;
    protected boolean vertexBeingDragged;

    public PolygonTool(DrawCanvas canvas) {
        super(canvas);
    }

    public void init() {
        super.init();
        canvas.setCursor(Cursor.getPredefinedCursor(
            Cursor.CROSSHAIR_CURSOR));
        iter = null;
        Geometry geom = manager.selected();
        if (!(geom instanceof DynamicPolygonGeometry))
            manager.select(null);
        else {
            DynamicPolygonGeometry poly =
                (DynamicPolygonGeometry)manager.selected();
            iter = poly.iterator();
        }
        manager.updateManager();
    }

    public void finish() {

```

```

        if ((iter != null) && (iter.nbrVertices() <= 2))
            removePolygon();
    }

    public HilightStrategy makeHilightStrategy() {
        return new PolygonHilightStrategy(this);
    }

    public void mousePressed(MouseEvent e) {
        vertexBeingDragged = findVertex(e.getX(), e.getY());
    }

    public void mouseReleased(MouseEvent e) {
        if (vertexBeingDragged && e.isControlDown())
            removeVertex();
        else if (!vertexBeingDragged) {
            PointGeometry p = new PointGeometry(e.getX(), e.getY());
            insertNewVertex(p);
        }
        vertexBeingDragged = false;
        manager.updateManager();
    }

    public void mouseDragged(MouseEvent e) {
        if (vertexBeingDragged) {
            moveVertex(e.getX(), e.getY());
            manager.updateManager();
        }
    }

    protected boolean findVertex(int x, int y) {
        if (iter == null)
            return false;
        PointZoneGeometry disk = new PointZoneGeometry(x, y);
        for (int i=0; i<iter.nbrVertices(); iter.next(), i++){
            PointGeometry p = iter.point();
            if (disk.contains(p))
                return true;
        }
        return false;
    }

    protected void insertNewVertex(PointGeometry p) {
        if (manager.selected() == null)
            makeNewPolygon(p);
        else
            iter.insertAfter(p);
    }

    protected void removeVertex() {
        if (iter.nbrVertices() == 1)
            removePolygon();
        else
            iter.remove();
    }

    protected void moveVertex(int x, int y) {
        iter.moveTo(x, y);
    }

    protected void makeNewPolygon(PointGeometry p) {
        Geometry poly = new DynamicPolygonGeometry(p);
        manager.add(poly);
        manager.select(poly);
        init();
    }

    protected void removePolygon() {
        Geometry poly = manager.selected();
        manager.remove(poly);
    }
}

```

```

    init();
}

protected PointGeometry selectedVertexPosition() {
    if (iter != null) return iter.point();
    else return null;
}
}

```

`PolygonTool`类定义了由高亮度显示策略使用的方法`selectedVertexPosition`。当多边形被高亮度显示时，策略调用这一方法来获得当前顶点的位置。

练习

7.18 实现`EllipseTool`类。

7.19 实际上，`DrawPad`程序包含有两种类型的工具：一种是画图工具（包括矩形、椭圆和多边形），另一种是用于指针工具（pointer tool）。请看是否有些行为是画图工具所共有而非画图工具没有？假如有一个类`AbstractDrawingTool`是所有画图工具的父型，请你确定图7-18的类图如何修改才能和这一新类相符？随着`DrawPad`程序的不断分析，你是不是能发现更多的`AbstractDrawingTool`的行为？

7.7.3 DrawPad的高亮度显示策略

前面讨论过，当`DrawPad`程序中的画布上的图形要刷新时，它交给图形管理器去完成。图形管理器中用一个组节点表示所有的图形，这个组节点包括（如果有）描述被选中图形的高亮度的子节点。图形管理器利用高亮度显示策略对不同的图形类型产生不同的显示效果。它调用高亮度显示策略的方法`makeHilight`，该方法返回描述高亮度的节点，然后图形管理器将该节点加入到组节点中。

可以有不同的产生高亮度的策略。当使用指针工具时，选中的图形的边线为红色粗线；当使用矩形工具时，选中的图形（矩形或椭圆）加上一个矩形定界框；当使用多边形工具时，选中的图形（多边形）的边线变亮，并且当前选中点为彩色亮点。

图形管理器并不实现产生高亮度的控制逻辑，而是交给高亮度显示策略完成。每个策略都实现`HilightStrategy`接口。这个接口中定义了抽象方法`makeHilight`用来产生不同的策略。图7-20中，三种策略都以自己的方式实现`makeHilight`方法。

当工具同画布一起安装时，工具分配一个合适的高亮度显示策略给图形管理器。为了做到这一点，每个工具都实现`makeHilightStrategy`方法，以便它返回所需的策略。当一个工具被安装时，它的`init`方法分配给图形管理器一个`makeHilightStrategy`返回的策略。举例来说明整个高亮度显示策略的应用过程。当用户点击`Pointer`按钮时，`PointerTool`的`init`方法调用`makeHilightStrategy`，`makeHilightStrategy`中产生`OutlineStrategy`对象并把它传给到图形管理器。这种设计使用策略设计模式来产生高亮度，使用工厂方法模式产生高亮度显示策略。

下面看一下`HilightStrategy`和它的子型的实现，其中方法`makeHilight`传入的是图形`g`，返回一个表示`g`的高亮度的节点。`HilightStrategy`接口定义了一个静态的三个像素的红色绘图工具。接口`HilightStrategy`定义如下：

```

public interface HilightStrategy {
    public final static Painter DefaultHilightPainter =

```

```

new DrawDynamicPolygonPainter(Color.red,
                               new BasicStroke(3));

public Node makeHilight(Geometry g)
    throws ClassCastException;
// EFFECTS: If g is null returns null; else if g's
// type is not compatible with this strategy
// throws ClassCastException; else returns
// a node representing a highlight for g.
}

```

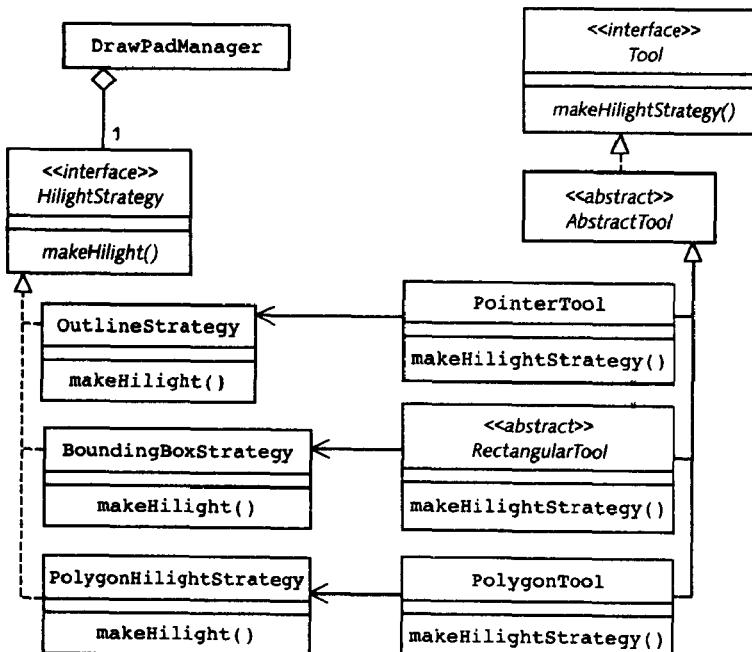


图7-20 高亮度显示中工厂方法和策略设计模式的应用

`OutlineStrategy`类把输入几何图形g的轮廓高亮度显示。由它继承的`DefaultHilightPainter`决定高亮度显示的外观。类`OutlineStrategy`的定义：

```

public class OutlineStrategy implements HilightStrategy {
    public Node makeHilight(Geometry g)
        throws ClassCastException {
        if (g == null)
            return null;
        else
            return new Figure(g, DefaultHilightPainter);
    }
}

```

`BoundingBoxStrategy`类在输入几何图形g的周围加上一个高亮度显示的定界框；如果输入图形不是矩形或椭圆，则`makeHilight`方法会产生异常。类`BoundingBoxStrategy`的定义：

```

public class BoundingBoxStrategy
    implements HilightStrategy {
    public Node makeHilight(Geometry g)
        throws ClassCastException {
        if (g == null)

```

```
        return null;
    else {
        RectangularGeometry r = (RectangularGeometry)g;
        return
            new Figure(r.boundingBox(),DefaultHilightPainter);
    }
}
```

`PolygonHilightStrategy`类把输入多边形g的轮廓高亮度显示，并且显示当前顶点。它定义了一个`tool`域，保存对`PolygonTool`的引用，通过它查询当前顶点的位置。类`PolygonHilightStrategy`的定义如下：

```
public class PolygonHilightStrategy
    implements HilightStrategy {

    public final static Painter DefaultVertexPainter =
        new FillPainter(Color.white);

    protected PolygonTool tool;

    public PolygonHilightStrategy(PolygonTool tool) {
        this.tool = tool;
    }

    public Node makeHilight(Geometry g)
        throws ClassCastException {
        if (g == null)
            return null;
        else {
            GroupNode node = new GroupNode();
            node.addChild(new Figure(g, DefaultHilightPainter));
            Geometry vertexPos = tool.selectedVertexPosition();
            if (vertexPos != null)
                node.addChild(
                    new Figure(vertexPos, DefaultVertexPainter));
            return node;
        }
    }
}
```

DrawPad程序的设计简化了增加新的形状按钮的过程。如果我们希望增加圆角矩形、点、线和曲线等几何图形按钮，只需遵循下列步骤即可：

- 如果以前没有定义过几何图形类，先需定义一个对应几何图形类，这个类实现 **AreaGeometry** 接口。
- 定义一个类来表示创建和修改几何图形实例，这个类实现 **Tool** 接口。
- 如果还没有合适的高亮度显示策略类，定义相应的类。
- 修改控制面板 **ControlPanel** 类以便：
 - (a) 增加一个 **ToolButton** 按钮到容器控制面板中。
 - (b) 调整 **ControlPanel** 的布局管理器，以恰当排列按钮的显示。

在下面的练习中，会有此类工作。

练习

7.20 在DrawPad程序的基础上，增加一个绘制圆角矩形的按钮（参考练习5.19中

RoundRectangleGeometry类的说明)。按钮的行为与矩形工具和椭圆形工具相似,圆角矩形角的长和宽都为50个像素。

7.21 修改练习7.20,使圆角矩形角的长和宽和它本身的长和宽成比例。例如,角长为它本身长的0.25倍加上20个像素,同样角宽为它本身宽的0.25倍加上20个像素。

[提示: RoundRectangleTool类需要覆盖UpdateRectangularGeometry方法。]

7.22 同PointZoneGeometry定义到某一点P的距离在R内的点区域相似(练习7.3),类LineSegmentZoneGeometry是定义到某一段线段L的距离在R内的线段区域(如果某一点到L的距离小于R,则称点P在线段区域中),R称为区域半径。你可以想象出线段区域和线段L的关系。LineSegmentZoneGeometry类实现AreaGeometry接口,下面给出它的框架:

```
public class LineSegmentZoneGeometry
    extends LineSegmentGeometry
    implements AreaGeometry {
public LineSegmentZoneGeometry(int x0, int y0,
                               int x1, int y1, int zoneRadius)
    throws IllegalArgumentException
// EFFECTS: If radius <= 0 throws
//   IllegalArgumentException; else constructs
//   the line segment (x0,y0)-(x1,y1) with
//   the specified zone radius.

public LineSegmentZoneGeometry(int x0, int y0,
                               int x1, int y1)
// EFFECTS: Constructs the line segment
//   (x0,y0)-(x1,y1) zone radius of 2 pixels.

public LineSegmentZoneGeometry(PointGeometry p0,
                               PointGeometry p1,
                               int radius)
    throws NullPointerException,
           IllegalArgumentException
// EFFECTS: If p0 or p1 is null throws
//   NullPointerException; else if radius <= 0
//   throws IllegalArgumentException; else
//   constructs the line segment from p0 to p1
//   with specified zone radius.

public LineSegmentZoneGeometry(PointGeometry p0,
                               PointGeometry p1)
    throws NullPointerException
// EFFECTS: If p0 or p1 is null throws
//   NullPointerException; else if radius <= 0
//   throws IllegalArgumentException; else
//   constructs the line segment from p0 to p1
//   with zone radius of two pixels.

public boolean contains(int x, int y)
// EFFECTS: Returns true if the point (x,y) is
//   no greater than zone radius pixels from
//   (some point in) this line segment;
//   else returns false.

public boolean contains(PointGeometry p)
    throws NullPointerException
```

```

// EFFECTS: If p is null throws
//   NullPointerException; else returns true
//   if point p lies no greater than zone
//   radius pixels from (some point in) this
//   line segment; else returns false.

public void setZoneRadius(int newR)
    throws IllegalArgumentException
// MODIFIES: this
// EFFECTS: If newR <= 0 throws
//   IllegalArgumentException; else sets
//   the zone radius to newR.

public int getZoneRadius()
// EFFECTS: Returns the current zone radius.
}

```

方法Line2D.ptSegDist返回一个输入点到一条线段的距离，这个方法用在下面的两个参数的contains方法的实现中，你可能想将该方法包括在你的类定义中：

```

// method of LineSegmentZoneGeometry class
public boolean contains(int x, int y) {
    Line2D shape = (Line2D)shape();
    return (shape.ptSegDist(x, y)<=getZoneRadius());
}

```

7.23 DrawPad程序增加一个线段绘制工具，下面是它的用法描述：

- 当用户点击该工具时，如果有选中图形并且为线段，则选中图形不变；反之，则选中图形被释放，并且光标变为十字图标。
- 当一条线段被选中时，用户可用鼠标选择拖动它的任一端点到任何新位置，它的另一个端点保持不变。
- 如果没有选中图形，用户用鼠标点击然后拖动鼠标到另一位置，则创建一条线段，起始点和结束点分别为两个端点。

线段选择时需借助于LineSegmentZoneGeometry对象来判断是否选中（离线段距离不大于区域半径R的点为有效选择点），其中区域半径为4比较合适。

7.24 DrawPad程序增加一个根据填充颜色设置图形轮廓颜色的控制按钮，功能同ControlPanel的ColorButton相似。在创建一个新图形时，它被赋予以当前填充颜色填充的绘图工具（painter）和具有当前轮廓颜色的stroke。

7.25 DrawPad程序增加一个像素宽度的复选框，复选框的项为数字串"1", "2", "3", "4"，并在组合框边上加一个标识组合框作用的标签。创建新图形时，它被赋予以当前填充色填充的绘图工具和具有当前轮廓颜色的stroke，宽度用上面选择的值。

小结

在某一特定应用领域中，面向对象应用程序框架是一种可重用的软件系统。无论是在应用程序领域还是程序设计方面，框架都可以使我们重用前人的智慧结晶。当使用框架开发应用程序时，开发者要定制该框架的类，这样设计出的类适用将来其他应用程序的使用或扩展。框架定制有两种实现方法：继承（白盒）和组合（黑盒）。

一个应用程序全部的设计都以它的框架为基础，通常这个设计暗示一种控制的颠倒：框架提供的程序调用应用程序提供的组件。尽管程序开发放弃了应用程序设计中的一些

控制，但使用框架仍然简化和方便于复杂、可靠软件的开发。虽然学习框架的使用要花费一定的时间和精力，但是与从头开发程序相比，还是付出的要少得多，何况框架的重用性可以使以后的项目开发受益更多。

AWT和Swing是Java图形界面程序开发的框架。Java框架主要由三个部分组成：组件、布局管理器和事件模型。组件是用户可视可交互的窗口小部件，比如按钮、面板、对话框、菜单、文本框和列表框等，容器是可以容纳其他组件的组件，布局管理器按一定的方式排列容器中的组件，而事件模型是组件事件的处理行为。

附录A 用户输入的读入和分析

本书中有相当多的程序用到从标准输入设备输入字符，典型的方法是从键盘输入。正是这个原因，我们特意设计了一个名为 `ScanInput` 的类，用来读入数字和字符串。先看一个使用这个类的简单例子：

```
public class TryScanInput {
    public static void main(String[] args) {
        try {
            ScanInput in = new ScanInput();
            System.out.print("Please enter your first name: ");
            String name = in.readString();
            System.out.print("Enter an integer: ");
            int a = in.readInt();
            System.out.print("Enter any number: ");
            float b = in.readFloat();
            String res =
                "Don't you know " + a + " * " + b + " is ";
            res += (a*b) + ", " + name + "!";
            System.out.println(res);
        } catch (NumberFormatException e) {
            System.out.println("I can multiply only numbers!");
        } catch (IOException e) {
            System.out.println("unexpected i/o exception");
        }
    }
}
```

下面是两个交互的例子，其中用户输入为黑体部分：

```
> java TryScanInput
Please enter your first name: Arianna
Enter an integer: 4
Enter any number: 2.5
Don't you know 4 * 2.5 is 10.0, Arianna!
> java TryScanInput
Please enter your first name: Phyllis
Enter an integer: 18
Enter any number: David
I can multiply only numbers!
```

`ScanInput` 类顺序从输入流中读取数据。按照约定，输入数据之间用空白字符分开，包括空格、跳格符和换行符。如果不按约定输入，没有在字符串之间加空白字符，多长的字符串都会当作一个字符串处理，例如，输入

```
thisIsARidiculousString
```

调用 `readString` 方法处理时，它会返回一个字符串 `thisIsARidiculousString`，尽管在英文里这是五个单词。同样，如果输入

6.3-7.42

调用 `readFloat` 方法会返回错误信息，因为它不是一个数字。相反，如果输入

6.3 -7.42

调用 `readFloat` 方法会读入 6.3，如果接着再一次调用 `readFloat`，读入 -7.42。下面是 `ScanInput` 类的说明：

```

public class ScanInput {

    public ScanInput(Reader inReader)
        // REQUIRES: inReader is not null.
        // EFFECTS: Constructs a new ScanInput whose input
        // comes from inReader.

    public ScanInput()
        // EFFECTS: Constructs a new ScanInput whose input
        // comes from the standard input System.in.

    public String readString() throws IOException
        // MODIFIES: this
        // EFFECTS: If i/o exception throws IOException;
        // else returns the next string in the input
        // stream (a string is a maximal-length sequence
        // of nonwhitespace characters); returns the empty
        // string if at end-of-file.

    public int readInt()
        throws IOException, NumberFormatException
        // MODIFIES: this
        // EFFECTS: If i/o exception throws IOException;
        // else if the next string in the input stream is
        // badly formed throws NumberFormatException; else
        // returns the integer the next string denotes.

    public double readDouble()
        throws IOException, NumberFormatException
        // MODIFIES: this
        // EFFECTS: If i/o exception throws IOException;
        // else if the next string in the input string is
        // badly formed throws NumberFormatException; else
        // returns the double that the next string denotes.

    public float readFloat()
        throws IOException, NumberFormatException
        // MODIFIES: this
        // EFFECTS: If i/o exception throws IOException;
        // else if the next string in the input stream is
        // badly formed throws NumberFormatException; else
        // returns the float that the next string denotes.

    public boolean eof()
        // EFFECTS: Returns true if at end-of-file;
        // else returns false.
}

```

`ScanInput` 有一个带参数的构造器，它可以指定任何类型的输入设备。比如，可以用下面方法把 `ScanInput` 和一个文件名 `filename` 联系起来，让它从文件中读入数据：

```

ScanInput in = new ScanInput(new FileReader("filename"));

ScanInput 类定义如下：

public class ScanInput {

    protected static final int MAX_STRING_LEN = 512;

```

```

protected Reader in;
protected boolean eof;
protected OneCharBuf buf;

public ScanInput(Reader inReader) {
    in = new BufferedReader(inReader);
    eof = false;
    buf = new OneCharBuf();
}

public ScanInput() {
    this(new InputStreamReader(System.in));
}

public String readString() throws IOException {
    char[] buf = new char[MAX_STRING_LEN];
    char c;
    int count = 0;
    skipWhitespace();
    if (eof()) return new String();
    for (c = readChar(); !isWhitespace(c) && !eof();
         c = readChar())
        buf[count++] = (char)c;
    return new String(buf, 0, count);
}

public int readInt()
    throws NumberFormatException, IOException {
    String s = readString();
    return Integer.parseInt(s);
}

public double readDouble()
    throws NumberFormatException, IOException {
    String s = readString();
    return Double.parseDouble(s);
}

public float readFloat()
    throws NumberFormatException, IOException {
    return (float)readDouble();
}

public boolean eof() {
    return eof;
}

//
// protected methods
//
protected char readChar() throws IOException {
    char c = (char)0;
    int cint;
    if (buf.isFull())
        c = buf.get();
    else {
        cint = in.read();
        if (cint == -1) eof = true;
        else c = (char)cint;
    }
    return c;
}

```

```
protected void unreadChar(char c) {
    if (buf.isFull())
        throw new Error("internal error");
    else
        buf.set(c);
}

protected boolean isWhitespace(char c) {
    return Character.isWhitespace(c);
}

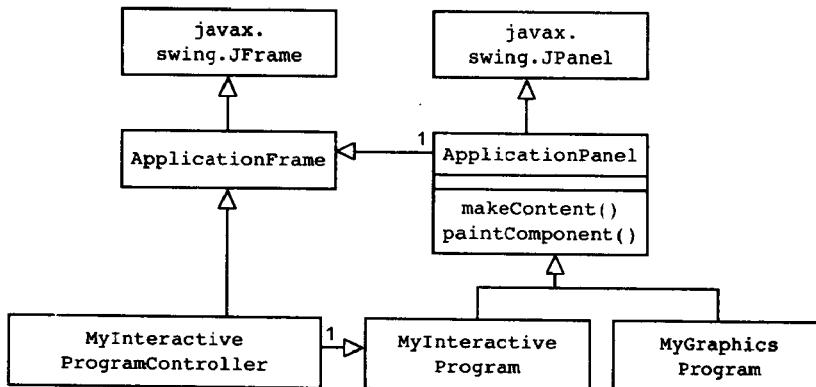
protected void skipWhitespace() throws IOException {
    char c = (char)0;
    do {
        c = readChar();
    } while (isWhitespace(c) && !eof());
    if (eof()) eof = true;
    else unreadChar(c);
}

// nested class for managing one-character buffer
static class OneCharBuf {
    char c;
    boolean isFull = false;
    void set(char c) {
        this.c = c;
        isFull = true;
    }
    char get() {
        isFull = false;
        return c;
    }
    boolean isFull() {
        return isFull;
    }
}
```

附录B 图形程序框架

在第7章之前，本书的所有图形程序都基于两个不同的程序模板：静态图形程序模板MyGraphicsProgram(3.5节)和用户可交互的动态图形程序模板MyInteractiveProgram(4.5节)。无论使用哪种模板，它们都要求编程人员定义一个类来扩展ApplicationPanel类。使用静态图形程序，新类（在图B-1中为MyGraphicsProgram）需要覆盖父类的makeContent方法和paintComponent方法，以按自己的要求创建和绘制图形内容。

使用动态交互图形程序，新类（在图B-1中为MyInteractiveProgram）要覆盖父类的paintComponent方法。此外，还需定义一个辅助类，由它控制交互操作和维护图形内容（在图B-1中为MyGraphicsProgramController）。MyInteractiveProgram类把控制器作为它的组件。当需要刷新时，它向控制器发送消息，要求返回当前的图形内容，刷新还是由它自己完成；另外控制器保存框架的引用，当它发现有图形内容变化时，也会发送repaint消息给框架。



图B-1 图形程序框架

下面是ApplicationPanel类和ApplicationFrame类的定义：

```
public class ApplicationPanel
    extends javax.swing.JPanel {

    // the frame that owns this application
    protected ApplicationFrame frame;

    // to be overridden if the subclass takes
    // program arguments
    static public void parseArgs(String[] args) {
        // EFFECTS: Processes program arguments, if any.
    }

    public void makeContent() {
        // MODIFIES: this
        // EFFECTS: Constructs the graphics content.
    }
}
```

```

protected void setFrame(ApplicationFrame frame) {
    // MODIFIES: this
    // EFFECTS: Sets this panel's frame.
    this.frame = frame;
}

protected ApplicationFrame getFrame() {
    // EFFECTS: Gets this panel's frame.
    return this.frame;
}

public class ApplicationFrame extends JFrame {

    public static String DEFAULT_TITLE = "My Frame";
    public static int DEFAULT_WIDTH = 400;
    public static int DEFAULT_HEIGHT = 400;

    public ApplicationFrame(String title, int width,
                           int height) {
        // REQUIRES: title is non-null, and width and
        // height are positive.
        // EFFECTS: Constructs a new frame with given
        // title and size.
        super(title);
        setSize(width, height);
        center();
        addWindowListener(new WindowAdapter() {
            public void windowClosing(WindowEvent e) {
                dispose();
                System.exit(0);
            }
        });
    }

    public ApplicationFrame(String title) {
        // REQUIRES: title is non-null.
        // EFFECTS: Constructs a new frame of given title
        // and default size.
        this(title, DEFAULT_WIDTH, DEFAULT_HEIGHT);
    }

    public ApplicationFrame(int width, int height) {
        // REQUIRES: width and height are positive.
        // EFFECTS: Constructs a new frame of given size.
        this(DEFAULT_TITLE, width, height);
    }

    public ApplicationFrame() {
        // EFFECTS: Constructs a new frame of default size.
        this(DEFAULT_TITLE, DEFAULT_WIDTH, DEFAULT_HEIGHT);
    }

    public void center() {
        // EFFECTS: Centers this frame within the screen.
        Dimension screenSize =
            Toolkit.getDefaultToolkit().getScreenSize();
        Dimension frameSize = getSize();
        int x = (screenSize.width - frameSize.width) / 2;
        int y = (screenSize.height - frameSize.height) / 2;
        setLocation(x, y);
    }
}

```

```
}

public void setPanel(ApplicationPanel panel) {
    // MODIFIES: this
    // EFFECTS: Adds panel to this frame.
    Container contentPane = getContentPane();
    contentPane.add(panel);
    panel setFrame(this);
    panel.setPreferredSize(getContentSize());
    pack();
}

public Dimension getSize() {
    // EFFECTS: Returns the size of this frame's
    // content pane.
    Dimension d = getSize();
    Insets insets = getInsets();
    int w = d.width - insets.left - insets.right;
    int h = d.height - insets.top - insets.bottom;
    return new Dimension(w, h);
}
}
```

附录C 统一建模语言UML符号概述

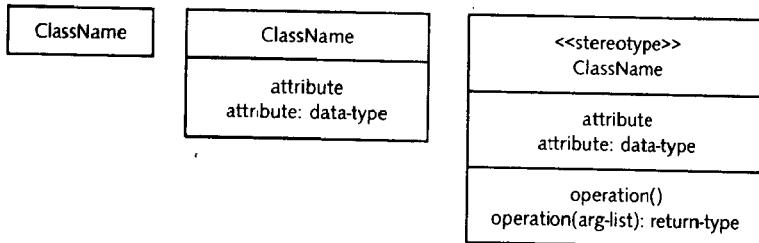
统一建模语言（Unified Modeling Language, UML）是面向对象建模的表示法。UML表示法是用各种不同图形定义表达设计中的各种视图。UML图形描述的是所设计系统的模型蓝图。它对充分理解系统、实现系统、不同的人员相互交流和系统文档化都是非常重要的，而且系统模型在整个系统分析和设计阶段是随着过程的不断推进而不断修改和完善的。自从1997年被OMG（Object Management Group）确定为面向对象建模的标准，UML获得了工业界、科技界和应用界的广泛支持，已有多个公司的联盟表示支持采用UML作为建模语言。

本书中只介绍UML的一小部分（参考文献中列出了几本学习UML的好书），主要是帮助了解系统设计两大相关部分：用于描述系统静态结构的类图（class diagram）和用于表达系统动态结构的顺序图（sequence diagram）。类图表示类和类之间、对象和对象之间的关系。所谓类是对一类具有相同特征的对象的描述，而对象则是系统运行时类的实例。

系统动态结构用于表达系统运行时的各种行为变化。顺序图用来描述对象之间动态的交互关系，着重体现对象间消息传递的时间顺序。

C.1 类图

类图表示类和类之间的关系。在UML中，类的最简单的表示是在一个封闭的框内加上类名。更详细一点，框内可以包括关键属性和关键方法，如果有必要这些方法还可以分类。类名的前面可以加上表示类的构造型，例如包括<<abstract>>或<<interface>>。抽象的元素（包括抽象类和抽象接口）表示为斜体。

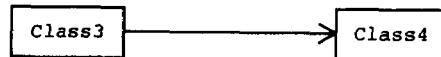


两个类之间存在某种语义上的联系称为关联（association），例如一个类的实例创建或发消息给另一个类的实例。关联是用连接两个类之间的一条线表示。关联是可以命名的，在名字边用小实心三角箭头表示命名的方向。例如，在下面的类图中，如果用`approves`替换`association name`就表示`Class1 approves Class2`。



关联是可以有方向的。关联上加上箭头表示方向，在UML中称为导航（navigability），表示类实例的消息发送方向。只在一个方向上存在导航称作单向关联，在两个方向上都

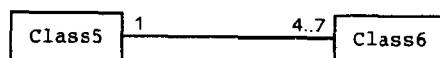
有导航称作双向关联。如果不带箭头导航就是未确定的或者双向的。下图单向关联表示 Class3 的实例可以发消息给 Class4，但相反却不行。



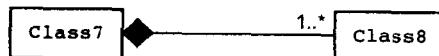
关联两头的类以某种角色参与关联，在关联线的末端加上名称表示角色名。如果在关联上没有标出角色名，则隐含地用类名作为角色名。角色还具有多重性 (multiplicity)。在关联线末端加上多重性值 (multiplicity value) (也称基数约束) 表示参与关系的对象的数目。下面解释多重性值的表示法 (下面出现的变量表示非负整数)：

n	表示 n
$a..b$	从 a 到 b ($a \leq b$)
a, b, c	表示 a 或 b 或 c
$n..*$	从 n 到多个
*	从 0 到多个

下图表示 Class5 实例可以与 4 到 7 个 Class6 的实例关联，相反动向， Class6 的实例只能与 1 个 Class5 的实例相关联。



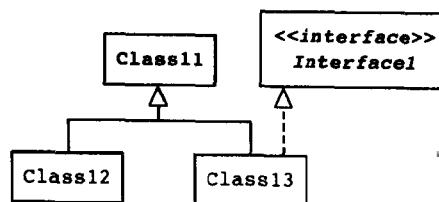
组合 (composition) 是一种特殊形式的关联，表示类之间的关系是整体与部分的关系。一个类的对象 (整体) 拥有另一个类的对象 (部分)。它们之间是整体拥有各部分，部分与整体共存的关系，如整体不存在了，部分也会随之消失。组合表示为实心菱形，实心菱形在组合体的这一端。下面表示 Class7 对象拥有 1 个或多个 Class8 对象。



聚集 (aggregation) 是另一种特殊形式的关联，也表示类之间的关系是整体与部分的关系。聚集有时被看作是弱组合，这是因为一个部分可以参加多个整体。聚集表示为空心菱形，空心菱形在整体的这一端。下面类图表示 Class9 对象包括 2 个 Class10 对象，同时表示 Class10 对象可以被任意个 Class9 对象共享。



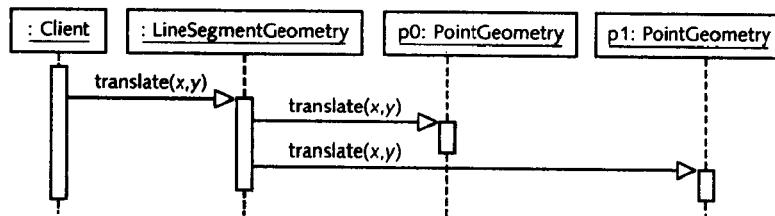
继承 (inheritance) 定义了一般元素和特殊元素之间的分类关系。继承表示为在父类端的连线头为空心三角形，实线表示扩展，虚线表示接口的实现。下图中类 Class12 扩展了类 Class11，而类 Class13 既扩展了类 Class11，又实现了接口 Interface1。



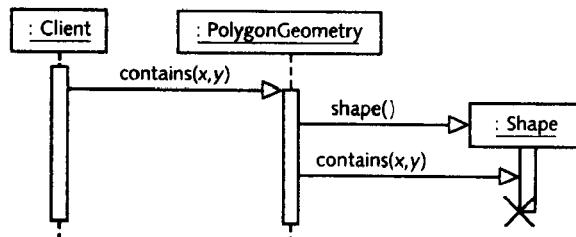
C.2 顺序图

顺序图用来描述在给定的场景中对象之间动态的交互关系，着重体现对象间消息传递的时间顺序。顺序图存在两个轴：水平轴表示不同的对象，垂直向下轴表示时间（自上而下）。顺序图中的对象用一个带有垂直虚线的矩形框表示，并标有对象的类名。垂直虚线是对象的生命线，用于表示在某段时间内对象是存在的。从发送对象生命线到接收对象生命线的带有箭头的水平线表示消息。

顺序图中对象是用带下划线的类名前面加上冒号表示的。如果有同一个类生成的两个或多个对象，而且需要区分它们，那么可以在冒号前加上对象名，即表示为：对象名：类名。对象是由发送给它的消息激活的。当收到消息时，接收对象立即开始执行活动，并有可能发送它产生的消息给自己或其他的对象；当完成它的工作时，对象停止活动。当然对象在下次还是可以被激活的。在对象生命线上显示一个细长竖形柱来表示对象被激活，竖形柱的长度覆盖了对象的活动期。如下图，当一个LineSegmentGeometry对象收到一个translate消息时，它就发送translate消息给对象p0和p1。

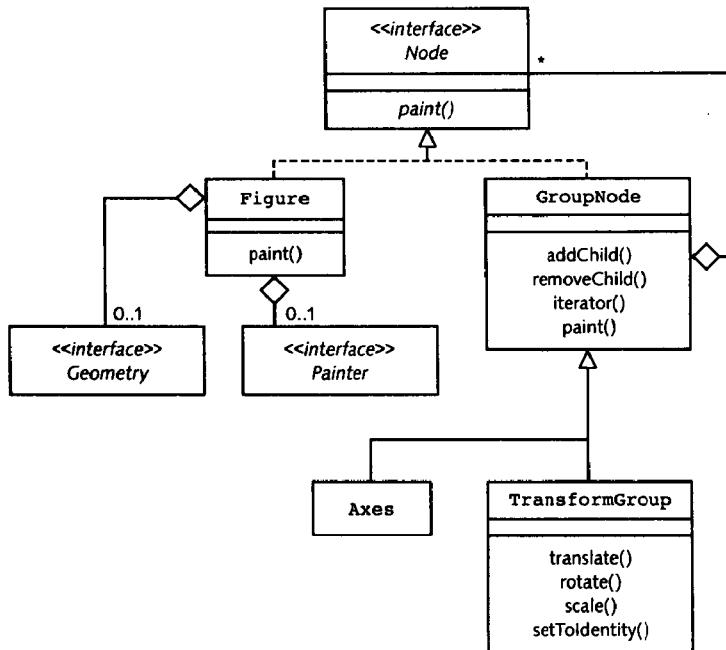


一个对象有时可以通过发送消息来创建或释放另一个对象。对象的创建是用接收消息末端的对象框表示的；当一个对象被释放或自我释放时，该对象用“X”标识。尽管Java的垃圾回收系统会在对象被释放后某个时候才将其所占资源回收，但是不再被引用的对象是要有效释放的。如下例所示，当多边形收到contains消息询问它是否包含点(x, y)时，它创建一个形状，然后发送消息contains给这一形状，询问是否包含点(x, y)。当形状回答了它是否包含输入点后，它就被释放。



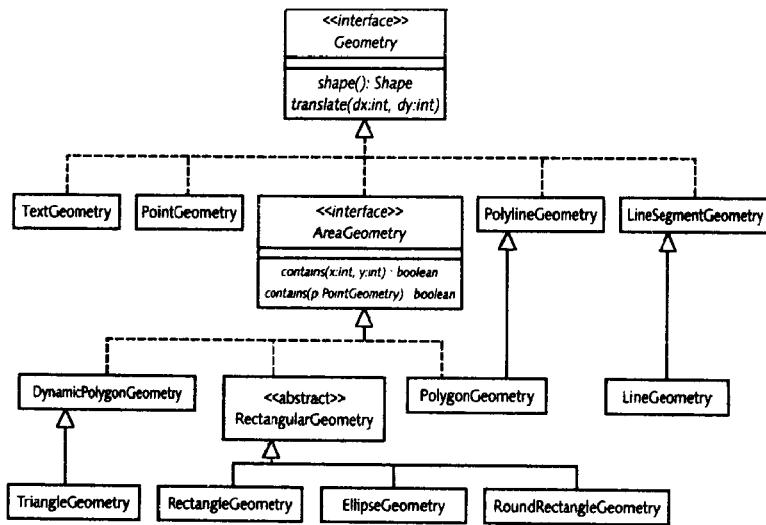
附录D banana包结构

本附录通过一系列的类图来描述banana包结构。它的最基本功能是用包中类创建表示组合图形的情景图。情景图用来描述类和它的组成元素之间的关系，情景图的元素称为节点。情景图的内部节点（带有子节点的节点）是组节点（group node），即 **GroupNode** 类或它的子类的实例。**TransformGroup** 节点也是一个组节点，它是在父节点坐标系的基础上定义了一个新的坐标系。**Axes** 类也是一个组节点，它创建了坐标系的一对坐标轴。情景图中叶节点是图形（Figure类的实例），它至多包括一个几何图形（geometry）和一个绘图工具（painter）。关于下面情景图的详细描述，请参见 6.4 节。

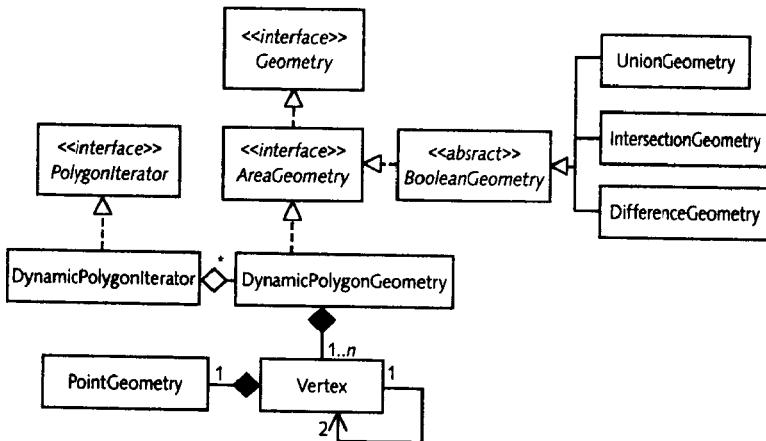


几何图形是图形（figure）中的两个重要组件之一。我们定义过很多的表示几何图形的类：**AreaGeometry** 类描述封闭区域几何图形，**RectangularGeometry** 类描述矩形几何图形（5.4.2 和 5.4.3 节）；**PolylineGeometry** 表示平面上连续直线段组成的折线（4.3.2 节），**PolygonGeometry** 表示平面上的多边形（5.3.1 节）；**DynamicPolygonGeometry** 表示可以修改的动态多边形，包括新增顶点、删除顶点和移动顶点（6.2.2 节）；**TriangleGeometry** 在 6.2 节的三角形计算中起很大作用；其他的还包括 **TextGeometry**（练习 5.20）、**PointGeometry**（3.2.1 节）、**LineSegmentGeometry**（练习 3.3）、**LineGeometry**（5.2.3 节）和各种不同的矩形几何图形包括 **RectangleGeometry**（3.2.2 节）、**EllipseGeometry**（4.4.2 节）和

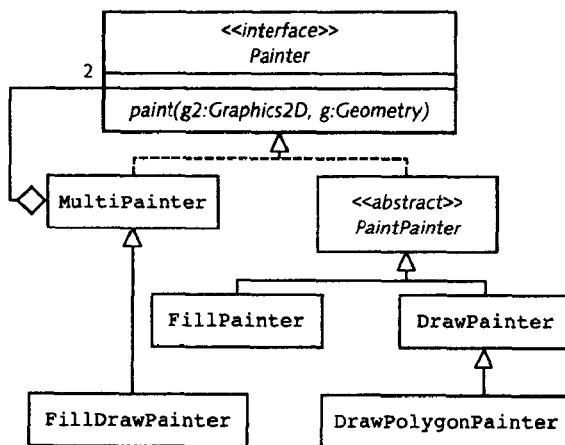
RoundRectangleGeometry (5.4.2节)。



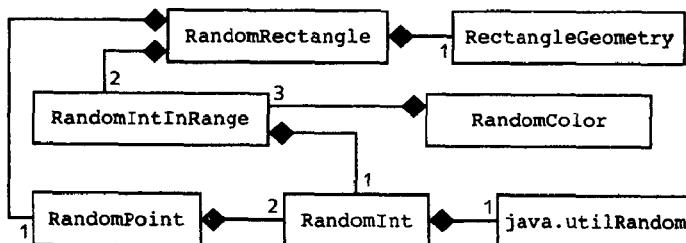
动态多边形由一个或多个顶点组成 (6.2.2节)。一个顶点表示为平面上的一个点和另外两个相关顶点 (它们分别是它的上一个顶点和下一个顶点)。客户使用多边形迭代器来维护动态多边形 (6.2.3节)。布尔几何图形是对两个几何图形进行并、交、差操作得到的新几何图形 (6.3.1节)。



图形由一个几何图形和一个绘图工具组成。在5.6节中, 我们详细描述了Painter类型, 并介绍了它的paint方法。不同的绘图工具决定图形的外观, 如用蓝色填充内部或用绿色画轮廓。抽象类PaintPainter实现了它的子类的paint特性; FillPainter和DrawPainter分别用来填充和画轮廓线; (其中DrawPainter不适用绘制轮廓线和形状不一致的动态多边形, 例如一个顶点的多边形)。DrawPolygonPainter和DrawDynamicPolygonPainter分别来画静态和动态多边形的轮廓线; MultiPainter用来把一组绘图工具组合起来, 产生特殊的效果。当它的两个组件都是PaintPainter时, 它表示由两个绘图工具组成的序列; 当它的其中一个组件为MultiPainter时, 它是有至少三个或以上的绘图工具序列。



在制作随机效果图形时，有时候随机数生成器非常有用。我们介绍了几个这样的随机生成类：`RandomInt`使用Java的`Random`类从给定范围内获得随机整数；`RandomIntInRange`从固定范围内获得随机整数；`RandomPoint`在固定范围区域中产生随机点；同样`RandomRectangle`在固定范围区域中产生随机长方形；`RandomColor`产生随机颜色。所有这些随机数生成器都在4.2节中介绍。



下面的类层次对照列表列出了banana包中所有的类和接口。它们分别与介绍它的那个章节和练习相对照。缩进行表示它是上一行类的扩展。例如：`DifferenceGeometry`类扩展了`BooleanGeometry`类。由于banana包中的类包含在所有的章节中，所以有些类的父类是在它们介绍之后才产生的，例如，`PointGeometry`类实际上实现了`java.lang.Comparable`和`Geometry`两个接口，但在3.2.1节中介绍它时没有按实现这两个接口方法做。

类层次列表

Assert类	2.2.1节
Attribute类	练习3.4
BooleanGeometry类（实现AreaGeometry）	6.3.1节
DifferenceGeometry类	练习.6.19
IntersectionGeometry类	6.3.1节
UnionGeometry类	练习6.19
Dictionary类	练习4.21
DynamicPolygonGeometry类（实现AreaGeometry）	6.2.2节

RegularPolygonGeometry类	练习 6.11
TriangleGeometry类	练习 6.13
DynamicPolygonIterator类（实现PolygonIterator）	6.2.3 节
DynamicPolygons类	练习 6.9
Figure类（实现Node）	5.6.1 节
GroupNode类（实现Node）	6.4.1 节
Axes类	6.4.2 节
TransformGroup类	6.4.3 节
LineSegmentGeometry类（实现Geometry）	练习 3.3
LineGeometry类	5.2.3 节
LineSegmentZoneGeometry类（实现AreaGeometry）	练习 7.21
MultiDrawPainter类（实现Painter）	练习 5.35
MultiPainter类（实现Painter）	5.6.3 节
FillDrawPainter类	练习 5.34
PaintPainter类（实现Painter）	6.2.2 节
DrawPainter类	5.6.2 节
DrawDynamicPolygonPainter类	练习 6.8
DrawPolygonPainter类	5.6.4 节
FillPainter类	5.6.2 节
PointGeometry类（实现java.lang.Comparable, Geometry）	3.2.1 节
PointZoneGeometry类（实现AreaGeometry）	练习 7.3
TransformablePointGeometry类	5.2.2 节
PolylineGeometry类（实现Geometry）	4.3.2 节
PolygonGeometry类（实现AreaGeometry）	5.3.1 节
RandomColor类	练习 4.14
RandomInt类	4.2.2 节
RandomIntInRange类	4.2.3 节
RandomPoint类	4.2.4 节
RandomPointInRectangle类	练习 4.13
RandomRectangle类	4.2.5 节
Range类	练习 3.2
Rational类（实现java.lang.Comparable）	4.4.3 节
RectangularGeometry类（实现AreaGeometry）	5.4.2 节
EllipseGeometry类	4.4.2 节
RectangleGeometry类	3.2.2 节
RoundRectangleGeometry类	练习 5.19
ScanInput类	练习 3.2, 附录A
Sort类	5.5.2 节
TextGeometry类（实现Geometry）	练习 5.20

java.lang.Throwable类（实现java.io.Serializable）

java.lang.Exception类

java.lang.RuntimeException类

FailedConditionException类 2.2.1节

ZeroArraySizeException类 2.3节

接口层次列表

Geometry 接口	5.4.3节
AreaGeometry 接口	5.4.3节
Node 接口	6.4节
Painter 接口	5.6节
PolygonInterator 接口	6.2.3节

参考文献

Abelson, Harold, Gerald Sussman, and Julie Sussman. 1997. *Structure and Interpretation of Computer Programs*. 2nd ed. New York: McGraw-Hill.

Arnold, Ken, James Gosling, and David Holmes. 2000. *The Java Programming Language*. 3rd ed. Boston: Addison Wesley.

Bailey, Duane. 2000. *Java Structures: Data Structures in Java for the Principled Programmer*. Boston: McGraw-Hill.

Booch, Grady, James Rumbaugh, and Ivar Jacobson. 1999. *The Unified Modeling Language User Guide*. Reading, MA: Addison Wesley.

Budd, Timothy. 1999. *Understanding Object-Oriented Programming with Java*. Reading, MA: Addison Wesley.

Foley, James, Andries van Dam, Steven Feiner, John Hughes, and Richard Phillips. 1994. *Introduction to Computer Graphics*. Reading, MA: Addison Wesley.

Friedman, Daniel, Mitchell Wand, and Christopher Haynes. 2001. *Essentials of Programming Languages*. 2nd ed. Cambridge, MA: MIT press.

Gamma, Erich, Richard Helm, Ralph Johnson, and John Vlissides. 1995. *Design Patterns: Elements of Reusable Object-Oriented Software*. Reading, MA: Addison Wesley.

Geary, David. 1998. *Graphic Java: Mastering the JFC*. 3rd ed. Upper Saddle River, NJ: Prentice Hall.

Hardy, Vincent. 1999. *Java 2D API Graphics*. Upper Saddle River, NJ: Prentice Hall.

Horstmann, Cay, and Gary Cornell. 2000. *Core Java 2 Vol. 1: Fundamentals*. Upper Saddle River, Prentice Hall.

Jia, Xiaoping. 2000. *Object-Oriented Software Development Using Java: Principles, Patterns, and Frameworks*. Reading, MA: Addison Wesley.

Liskov, Barbara, with John Guttag. 2001. *Program Development in Java: Abstraction, Specification, and Object-Oriented Design*. Reading, MA: Addison Wesley.

Meyer, Bertrand. 1997. *Object-Oriented Software Construction*. 2nd ed. Upper Saddle River, NJ: Prentice Hall.

Pooley, Rob, and Perdita Stevens. 1999. *Using UML: Software Engineering with Objects and Components*. Harlow, England: Addison Wesley.

Richter, Charles. 1999. *Designing Flexible Object-Oriented Systems with UML*. Indianapolis, IN: Macmillan Technical Publishing.

Rumbaugh, James, Ivar Jacobson, and Grady Booch. 1999. *The Unified Modeling Language Reference Manual*. Reading, MA: Addison Wesley.

Vlissides, John. 1998. *Hatching Patterns: Design Patterns Applied*. Reading, MA: Addison Wesley.